Binary Trees and Parallel Scheduling Algorithms

ELIEZER DEKEL AND SARTAJ SAHNI

Abstract—**This paper examines the use of binary trees in the design of efficient parallel algorithms. Using binary trees, we develop efficient algorithms for several scheduling problems. The shared memory model for parallel computation is used. Our success in using binary trees for parallel computations, indicates that the binary tree is an important and useful design tool for parallel algorithms.**

Index Terms—**Complexity, design methodologies, parallel algorithms, scheduling, shared memory model.**

I. Introduction

Algorithm design techniques for single processor computers have been extensively studied. For example Horowitz and Sahni [15] extoll the virtues of such design methods as divide-and-conquer, dynamic programming, greedy method, backtracking, and branch-and-bound. These methods generally lead to efficient sequential (i.e., single processor) algorithms for a variety of problems. These algorithms, however, are not very efficient for computers with a very large number of processors. In this paper, we propose a design method that we have found useful in the design of algorithms for computers that have many processors. The method proposed here is called the *binary tree method.* While this method has been used in the design of parallel algorithms earlier, here we attempt to show its broad applicability to the design of such algorithms. It is hoped that further research will bring to light some other basic design tools for parallel algorithms. One should note that trees have been used extensively in the design of efficient sequential algorithms. In fact, divide-and-conquer, backtracking, and branch-and-bound all use an underlying computation tree [15]. The use of binary trees as proposed here is quite different from the use of trees in sequential computation.

With the continuing dramatic decline in the cost of hardware, it is becoming feasible to build computers with thousands of processors economically. In fact, Batcher [5] describes a computer (MPP) with 16 384 processors that is currently being built for NASA. In coming years, one can expect to see computers with a hundred thousand or even a million processing elements. This expectation has motivated the study of parallel algorithms. Since the complexity of a parallel algorithm depends very much on the architecture of the parallel computer on which it is run, it is necessary to keep the architecture in mind when designing the algorithm. Several parallel architectures have been proposed and studied. In this paper, we shall deal directly only with the single instruction stream, multiple data stream (SIMD) model. Our technique and algorithms readily adapt to the other models (e.g., multiple instruction stream multiple data stream (MIMD) and data flow models). SIMD computers have the following characteristics:

1) They consist of p processing elements (PE's). The PE's are indexed $0, 1, \cdots, p-1$ and an individual PE may be referenced as in PE(i). Each PE is capable of performing the standard arithmetic and logical operations. In addition, each PE knows its index.
2) Each PE has some local memory.
3) The PE's are synchronized and operate under the control of a single instruction stream.
4) An enable/disable mask can be used to select a subset of the PE's that are to perform an instruction. Only the enabled PE's will perform the instruction. The remaining PE's will be idle. All enabled PE's execute the same instruction. The set of enabled PE's can change from instruction to instruction.

While several SIMD models have been proposed and studied, in this paper we shall be primarily concerned with only the shared memory model (SMM). In the shared memory model, there is a common memory available to each PE. Data may be transmitted from PE(i) to PE(j) by simply having PE(i) write the data into the common memory and then letting PE(j) read it. Thus, in this model it takes only 0(1) time for one PE to communicate with another PE. Two PE's are not permitted to write into the same word of common memory simultaneously. The PE's may or may not be allowed to simultaneously read the same word of common memory. If the former is the case, then we shall say that *read conflicts* are permitted. In a realistic situation, the common memory will be divided into blocks of size q and a read conflict occurs whenever two PE's attempt to simultaneously access the same block. Throughout this paper, we assume that $q = 1$.

Most algorithmic studies of parallel computation have been based on the SMM. Agerwala and Lint [1], Arjomandi [2], Csanky [8], Eckstein [11], and Hirschberg [12] have developed algorithms for certain matrix and graph problems using the SMM. Hirschberg [13], Muller and Preparata [24], and Preparata [30] have considered the sorting problem for SMM. The evaluation of polynomials on the SMM has been studied by Munro and Paterson [25], while arithmetic expression evaluation has been considered by Brent [7] and others.

The mesh connected computer (MCC), cube connected computer (CCC), and perfect shuffle connected computer (PSC) are three other SIMD models. Efficient algorithms to sort and perform data permutations on an MCC can be found in Thompson and Kung [38], Nassimi and Sahni [26] and [27], and Thompson [37]. Thompson's algorithms [37] can also be used to perform permutations on a CCC and a PSC. Lang [19], and Lang and Stone [20], and Stone [36] show how certain permutations may be performed using shuffles and exchanges. Nassimi and Sahni [28] develop fast sorting and permutation algorithms for a CCC and PSC. Dekel, Nassimi, and Sahni [9] present efficient matrix multiplication and graph algorithms for CCC's and PSC's.

The algorithms considered in this paper are described explicitly only for the SMM. The algorithms are readily translated into algorithms for the other SIMD models. In some cases, it may be necessary to use the data broadcasting algorithms developed by Nassimi and Sahni [29] to accomplish this adaptation to the other models.

Throughout this paper, we assume that no read conflicts are allowed.

II. The Binary Tree Method

In the binary tree method, we make use of a binary computation tree. Such a tree is generally a complete binary tree. Fig. 1 shows such a tree with 11 leaf nodes. The nodes of this tree have been indexed (indexes appear outside each node) using the standard indexing scheme for complete binary trees.

With each node of the computation tree, we associate k, $k \geq 1$ subproblems. The computation consists of k passes over this computation tree. In pass 1, we proceed from the leaves to the root solving the first subproblem associated with each node; in pass 2, we proceed from the root to the leaves solving the second subproblem associated with each node; and so on. Every odd pass is from the leaves to the root while every even pass is from the root to the leaves.

As far as we are aware, the binary tree method has thus far been used only with $k = 1$. A simple example is finding the sum $\sum_{i=1}^{n} A(i)$ of n numbers. The computation tree used has n leaf nodes. The subproblem associated with each node is that of finding the sum of all the numbers represented by the leaves in the subtree of which it is a root. Fig. 1 shows the computation for the case $n = 11$.

Manuscript received October 4, 1980; revised July 13, 1982. This work was supported in part by the National Science Foundation under Grant MCS80-005856 and in part by the Office of Naval Research under Contract N000145-80-C-0650.

E. Dekel is with the Program in Computer Science, University of Texas at Dallas, Richardson, TX 75080.

S. Sahni is with the University of Minnesota, Minneapolis, MN 55455.

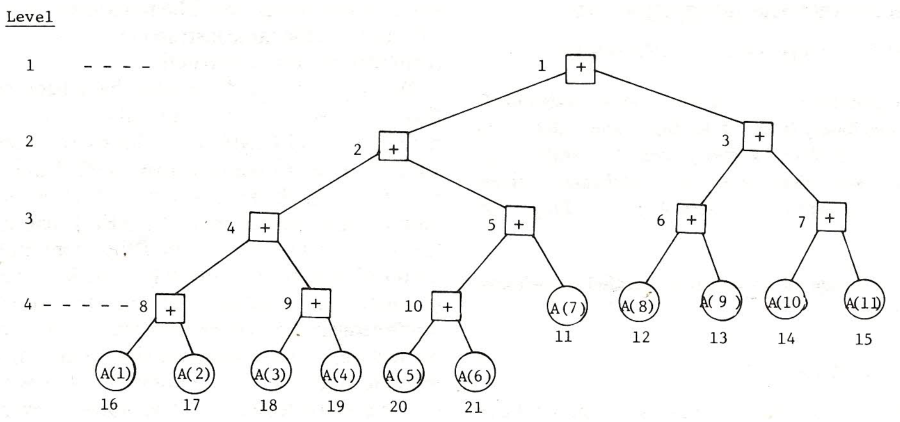


Fig. 1. Computation tree for $\sum_{i=1}^{n} A(i)$.

```
line procedure SUM1 $(A, n)$
    //compute $\sum_{i=i}^{n} A(i)$ using $\lfloor n/2 \rfloor$PE's//
    //initialize $V$//
1   $k \leftarrow \lfloor \log_2 n \rfloor; j \leftarrow 2^k; t \leftarrow 2*(n - j); p \leftarrow n - 1$
2   $V(p + i) \leftarrow A((i + t - 1) \bmod n + 1), 1 \le i \le n$
3   for $i \leftarrow k$ down to 0 do //add by levels//
4       $V(j) \leftarrow V(2j) + V(2j + 1), 2^i \le j \le \min\{p, 2^{i+1} - 1\}$
5   end for
6   return $(V(1))$
7   end SUM1
```

Fig. 2.

Once the computation has been described using a computation tree, a parallel algorithm is easily obtained. Let $V(i)$ denote the sum corresponding to node i. Procedure SUM1 of Fig. 2 is the corresponding parallel algorithm. In lines 2 and 4, the use of $a \le b \le c$ means that this line is to be executed in parallel for all b satisfying the inequality. Line 2 can be performed in two steps using $\lfloor n/2 \rfloor$ PE's. Line 4 needs at most $\lfloor n/2 \rfloor$ PE's. It is clear that the complexity of procedure SUM1 is $O(\log n)$.

In addition to analyzing the complexity of a parallel algorithm, one often (see Savage [32]) also computes the effectiveness of processor utilization (EPU). This is defined relative to a specific problem P, the complexity of the fastest sequential algorithm known for P, and the parallel algorithm A for problem P.

$$\text{EPU}(P, A) = \frac{\text{complexity of the fastest sequential algorithm for } P}{\text{number of PE's used by } A * \text{ complexity of } A}.$$

For the case of procedure SUM1,

$$\text{EPU} = \Omega\left(\frac{n}{n/2 * \log n}\right) = \Omega\left(\frac{1}{\log n}\right)$$

Note that $0 \le \text{EPU} \le 1$ and that an EPU close to 1 is considered "good." For the case of computing $\sum_{i=1}^{n} A(i)$, we can actually arrive at an $O(\log n)$ algorithm with an EPU of $\Omega(1)$ (i.e., using only $\lceil n/\log n \rceil$ PE's) [32]. This is done by dividing the $nA(i)$'s into $\lceil n/\log n \rceil$ groups, each group containing at most $\lceil \log n \rceil$ of the $A(i)$'s. Each of these groups is assigned to a PE which sequentially computes the sum of the numbers in the group. This takes $O(\log n)$ time. Now, we need to sum up these $\lceil n/\log n \rceil$ group sums. Procedure SUM1 can be used to compute this sum in $O(\log n)$ time.

Note that the discussion carried out so far concerning the computation of $\sum_{i=1}^{n} A(i)$ applies just as well to the computation of $\oplus_{i=1}^{n} A(i)$ where $\oplus$ is any associative operator (for example, max, min, *, etc.). Hence, $\max_{1 \le i \le n} \{A(i)\}$; $\min_{1 \le i \le n} \{A(i)\}$; $\prod_{i=1}^{n} A(i)$; etc., can all be computed in $O(\log n)$ time using $\lceil n/\log n \rceil$ PE's.

We now consider an example with $k = 2$. Suppose we wish to compute $S_j = \sum_{i=1}^{j} A(i)$, $1 \le j \le n$. We shall refer to this problem as the *partial sums problem.* When computing S_n using the straightforward sequential algorithm, we obtain S_i, $1 \le i \le n$ as a byproduct and so, in this case, no additional effort need be expended. In the case of procedure SUM1 (and its refinement to the case of $\lceil n/\log n \rceil$ PE's), however, all the S_i's are not computed during the computation of S_n. Following the computation of S_n, the remaining S_i's can be obtained by making one pass down the binary tree. In this pass each node transmits to its children the sum of the values to the left of the child.

Let $A(1{:}11) = (1, 1, 2, 3, 1, 2, 1, 2, 3, 4, 2)$. The computation tree of Fig. 1 is redrawn in Fig. 3. The index of each node appears outside it. Inside each node there are two numbers. The upper number is V as defined for procedure SUM1. The lower number in each node is L; where for any node i, L is defined as

$$L(i) = \begin{cases} 0 & i = 1 \\ L(i/2) & i \text{ is even} \\ L(i/2) + V(i - 1) & i \text{ is odd.} \end{cases}$$

One may easily verify that if i is a leaf node representing $A(j)$; then $L(i) = \sum_{1 \le p < j} A(p)$. Hence, from the L values of the leaf nodes, one can easily obtain all the partial sums. Our first algorithm for the partial sums problem is PSUM1 (Fig. 4). This algorithm simply computes the $V(i)$'s in the first pass and the $L(i)$'s in the second. Finally, the S values are computed.

As in the case of SUM1, the parallelism of lines 4 and 8 requires only $n/2$ PE's. Using $n/2$ PE's, line 2 can be done in two steps. Actually, procedure PSUM1 can be run in $O(\log n)$ time using only $\lceil n/\log n \rceil$ PE's. The idea here, is the same as that for SUM1.

By using a slightly different computation tree and rearranging the order of computation, one can arrive at a one pass algorithm for the partial sums problem. Let $A(0{:}n - 1)$ be the n numbers to be added. Let $S(0{:}n - 1)$ denote the partial sum array. A 2^k block of array elements consists of all array elements whose indexes differ only in the

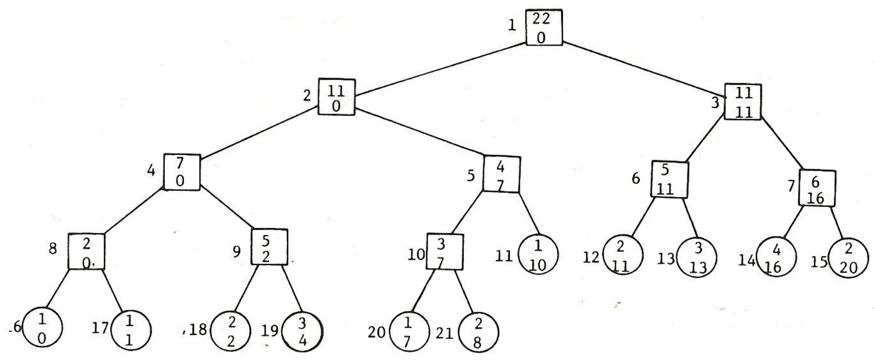


Fig. 3.

```
line procedure PSUM1 $(A, n, S)$
        //compute $S(i) = \sum_{j=1}^{i} A(j), 1 \le i \le n$//
1       $k \leftarrow \lfloor \log_2 n \rfloor; j \leftarrow 2^k; t \leftarrow 2*(n-j); p \leftarrow n-1$
2       $V(p+i) \leftarrow A((i+t-1) \bmod n+1), 1 \le i \le n$
3       for $i \leftarrow k$ down to 0 do //add by levels//
4           $V(j) \leftarrow V(2j) + V(2j+1), 2^i \le j \le \min\{p, 2^{i+1}-1\}$
5       end for
        //compute $L$s//
6       $L(1) \leftarrow 0$
7       for $i \leftarrow 1$ to $k+1$ do //compute $L$ by levels//
8           $L(j) \leftarrow$ if $j$ even then $L(j/2)$
                          else $L(j/2) + V(j-1)$
            endif
            $2^i \le j \le \min\{n, 2^{i+1}-1\}$
9       end for
10      $S((i+t-1) \bmod n+1) \leftarrow L(p+i) + V(p+i), 1 \le i \le n$
11      end PSUM1
```

Fig. 4.

least significant k bits. The 2^1 blocks of $A(0{:}10)$ are [0, 1], [2, 3], [4, 5], [6, 7], [8, 9], and [10]; the 2^2 blocks are [0, 1, 2, 3], [4, 5, 6, 7], and [8, 9, 10], etc. Two 2^k blocks are *sibling blocks* if their union is a 2^{k+1} block. Thus, [0, 1] and [2, 3] are sibling blocks; so also are [0, 1, 2, 3] and [4, 5, 6, 7]. However, [2, 3] and [4, 5] are not sibling blocks. The one pass algorithm computes S by first computing the partial sums for all 2^0 blocks of A. In this case, $S(i) = A(i)$. Next, S is computed for all 2^1 blocks; then for all 2^2 blocks; $\cdots$; and finally for the single 2^q block where $q = \lceil \log_2 n \rceil$.

Let X and Y be two sibling 2^k blocks. Let X be the block containing all elements with bit k equal to 0. The union of X and Y is a 2^{k+1} block. Relative to this 2^{k+1} block, the S values for elements of X are the same as with respect to the corresponding 2^k block. The S values for elements in Y, however, change by the sum of the A elements corresponding to the 2^k block X. Fig. 5 gives the S values and 2^k blocks when S values are computed by blocks as described above. Blocks are enclosed in brackets.

The updating of S when going from one block size to the next is easily performed if we keep track of the sum of the $A(i)$'s in each 2^k block. For this purpose, we use an auxiliary array T. $T(i)$ for i in a given 2^k block (except possibly the rightmost 2^k block) is the sum of all the $A(i)$'s in that block. Before we can formally specify the partial sums algorithm, we need a processor assignment scheme. Fig. 5 shows a processor assignment scheme for our example. Processors are assigned only to compute the S values that change. Thus, when $k = 0$, PE(0) computes $S(1)$, PE(1) computes $S(3)$, PE(2) computes $S(5)$, and PE(4) computes $S(9)$. When $k = 3$, PE(0) computes $S(8)$, PE(1) computes $S(9)$, and PE(2) computes $S(10)$. PE's 3 and 4 are idle when $k = 3$. Let $\cdots i_3 i_2 i_1 i_0$ be the binary representation of i. The PE assignment rule is obtained by defining the function $f(i, j) = \cdots i_{j+1} i_j 0 i_{j-1} \cdots i_0$. For any k, PE(i) computes $S(f(i, k) + 2^k)$ provided that this index of S is no more than $n - 1$. The one pass partial sums algorithm is stated as procedure PSUM2 (Fig. 6). PSUM2 uses $\lfloor n/2 \rfloor$ PE's indexed 0 through $\lfloor n/2 \rfloor - 1$.

It should be easy to see that our earlier ideas regarding the use of only $\lceil n/\log n \rceil$ PE's carry over to the case of PSUM2. So, PSUM2 can be modified to obtain an $0(\log n)$ one pass algorithm using only $\lceil n/\log n \rceil$ PE's. For the modified algorithm, EPU = $\Omega(1)$.

It should be emphasized that while a computation tree might look like a tree connected computer, there is no implication that the parallel algorithms obtained using the binary tree method are best run on such a computer. In fact, the computation at each node might require several processors.

Finally, there is a similarity between divide-and-conquer algorithms and the binary tree method when $k = 2$. In divide-and-conquer algorithms too, there is an underlying computation tree (not necessarily

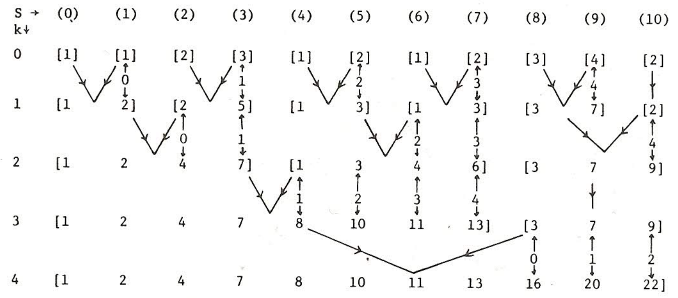


Fig. 5. Computing S by blocks.

binary). When stated recursively, the computation starts at the room, proceeds to the leaves, and then backs up to the root again. When stated iteratively, the computation starts at the leaves and proceeds to the root. Hence, iteratively stated divide-and-conquer algorithms are the same as binary tree method algorithms (extended to arbitrary trees) with $k = 1$.

In the rest of this paper, we demonstrate the usefulness of the binary tree method with $k = 2$ in arriving at efficient parallel algorithms for various scheduling problems.

III. Parallel Scheduling Algorithms

In this section, we develop fast parallel algorithms for a variety of scheduling problems. Each of these algorithms is arrived at using the binary tree method of Section I. We shall refrain from providing explicit formal statements such as those of Figs. 2, 4, and 6 of these algorithms. Instead, we shall describe the algorithms informally and illustrate them with an example. One should note that we are interested in both the complexity as well as the EPU of the algorithms developed.

All the scheduling problems to be discussed assume that n jobs have to be scheduled on m identical machines. Associated with job i is a four-tuple (r_i, d_i, p_i, w_i) where r_i is its *release time*, d_i is its *due time*, p_i is its *processing requirement*, and w_i is its weight, $1 \le i \le n$. The processing of no job can commence until its release time. No job can be scheduled for processing on more than one machine at any time instant. Job i is *completed* after it has been processed for p_i time units. If a job does not complete by its due time, it is *tardy*. In a *nonpreemptive schedule*, job i is scheduled to process on a single machine from some start time s_i to the completion time $s_i + p_i$, $1 \le i \le n$. In a *preemptive schedule* it is permissible to split the processing of jobs over machines as well as over nonadjacent time intervals.

A. Minimizing Maximum Lateness

Let S be a schedule for the n jobs (r_i, d_i, p_i, w_i). Let c_i be the completion time of job i. The *lateness* of job i is defined to be $c_i - d_i$. The maximum lateness, $L_{\max}$, is $\max_i \{c_i - d_i\}$. We wish to obtain an m machine nonpreemptive schedule that minimizes $L_{\max}$. This problem is known to be NP-hard [22]. So, we shall consider only special cases of this problem, i.e., cases for which a polynomial time sequential algorithm is known. Specifically, we shall consider the following cases: 1) $p_i = 1$, $1 \le i \le n$ and all release times are integer, 2) $m = 1$ (i.e., the number of machines is 1) and preemption is allowed, and 3) cases 1) and 2) with precedence constraints. These three cases are considered in Sections III-A1), III-A2), and III-A3), respectively. Since the weights w_i play no part in the $L_{\max}$ problem, we shall only consider triples (r_i, d_i, p_i) in these subsections.

1) $p_i = 1$, $1 \le i \le n$ and All Release Times are Integer: Jackson [16] has shown that when $m = 1$ and all jobs have the same release time, $L_{\max}$ is minimized by scheduling the jobs in nondecreasing order of due times. Horn [14] and Baker and Su [3] have generalized this method to the case where $m = 1$ and all jobs do not have the same release time. An optimal one machine schedule is now obtained by assigning jobs to time slots, one slot at a time starting at time 0. When we are considering the time slot $[i, i + 1]$, we select a job with least

```
line procedure PSUM2 $(A, S, n)$
     //one pass partial sums//
 1   declare $A(0{:}n-1), S(0{:}n-1), T(0{:}n-1)$
 2   for each PE($i$) do in parallel
     //initialize S and T for $2^0$-blocks//
 3   $j \leftarrow f(i, 0)$
 4   $S(j) \leftarrow T(j) \leftarrow A(j)$
 5   $S(j+1) \leftarrow T(j+1) \leftarrow A(j+1)$
 6   for $k \leftarrow 0$ to $\lceil \log_2 n \rceil - 1$ do
     //combine $2^k$-blocks//
 7     $j \leftarrow f(i, k)$
 8     if $j + 2^k \le n$ then
 9       $S(j + 2^k) \leftarrow S(j + 2^k) + T(j)$
10       $T(j + 2^k) \leftarrow T(j + 2^k) + T(j)$
11       $T(j) \leftarrow T(j + 2^k)$
12     endif
13   end for
14  end for
15  end PSUM2
```

Fig. 6. One pass partial sums algorithm.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r_i	5	2	2	5	2	2	6	2	5	6	9	9	9	9
d_i	8	7	7	10	3	6	7	5	12	17	16	11	15	16

(a)

i	5	8	6	2	3	1	4	9	7	10	12	13	11	14
r_i	2	2	2	2	2	5	5	5	6	6	9	9	9	9
d_i	3	5	6	7	7	8	10	12	7	17	11	15	16	16

(b)

Fig. 7.

due time from among the set of available jobs. (The set of available jobs consists of all jobs not yet selected that have a release time less than or equal to i.) If this set is empty, then this slot is left idle. This strategy can be implemented to run in $0(n \log n)$ time on a single processor computer. Blazewicz [6] has extended this idea to the general case $m \ge 1$. His algorithm also schedules by time slots. Let J be the set of jobs available when slot $[i, i + 1]$ is to be scheduled. If $|J| \le m$ then all the available jobs are processed in $[i, i + 1]$. If $|J| > m$, then we select m jobs with least due times.

In developing the parallel algorithm, we first consider the case $m = 1$. Horn's algorithm is readily seen to be highly sequential. No decision concerning time slot $[i, i+1]$ can be made unless we know the jobs that are available at this time. This, of course, depends on which jobs were selected for the earlier time slots. So, a straightforward adaptation of Horn's algorithm would need n steps (one for each time slot). The overall complexity of the resulting parallel algorithm would be $\Omega(n)$. This is not very good. We are really interested in algorithms with complexity $0(\log^k n)$ for some k.

Despite the highly sequential nature of Horn's method, his idea can be used to arrive at a parallel algorithm with complexity $0(\log^2 n)$. This is accomplished using the binary tree method. It is helpful to consider an example. Suppose we have 14 jobs with r_i, and d_i as specified in Fig. 7(a). The first step in our proposed parallel algorithm is to sort the jobs by release times (into nondecreasing order). Jobs with the same release time are sorted into nondecreasing order of due

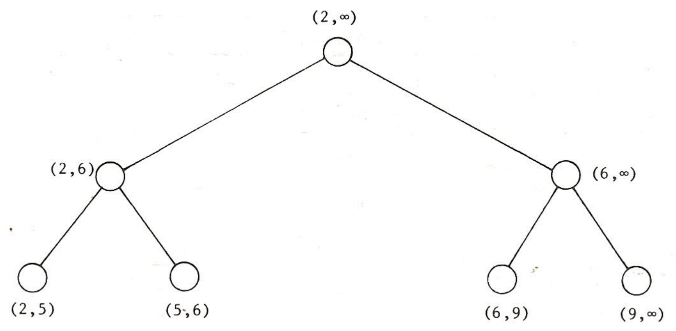
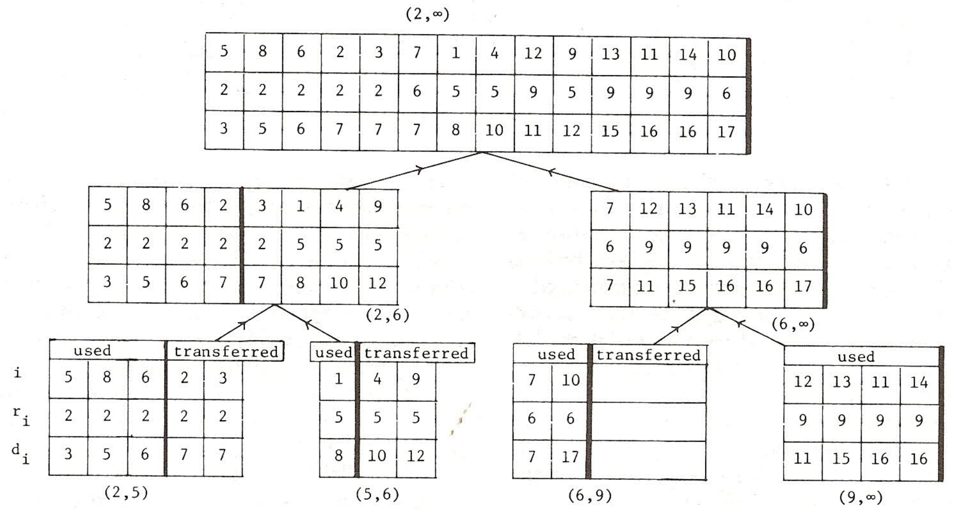


Fig. 8. Computation tree for the example of Fig. 7.



(2,∞)

5	8	6	2	3	7	1	4	12	9	13	11	14	10
2	2	2	2	2	6	5	5	9	5	9	9	9	6
3	5	6	7	7	7	8	10	11	12	15	16	16	17

(2,6)

5	8	6	2	3	1	4	9
2	2	2	2	2	5	5	5
3	5	6	7	7	8	10	12

(6,∞)

7	12	13	11	14	10
6	9	9	9	9	6
7	11	15	16	16	17

(2,5)

	used			transferred	
i	5	8	6	2	3
r_i	2	2	2	2	2
d_i	3	5	6	7	7

(5,6)

used	transferred	
1	4	9
5	5	5
8	10	12

(6,9)

used		transferred
7	10	
6	6	
7	17	

(9,∞)

used			
12	13	11	14
9	9	9	9
11	15	16	16

Fig. 9. First pass of the $L_{\max}$ algorithm.

time. Let $R_1, R_2, \cdots$, and R_k be the k distinct release times of the n jobs ($R_1 < R_2 < \cdots < R_k$). Let $R_{k+1} = \infty$. For our example, the sorted sequence of jobs is shown in Fig. 7(b); $k = 4$, and $R_1 = 2$, $R_2 = 5$, $R_3 = 6$, $R_4 = 9$, *and* $R_5 = \infty$.

Next, a binary computation tree is associated with the problem. The tree used is the unique complete binary tree with k leaves. With each node in this tree, we associate a time interval (t_L, t_R). Assume that the leaf nodes are numbered 1 through k, left to right. The ith leaf node has associated with it the interval (R_i, R_{i+1}), $1 \le i \le k$. The interval (t_L, t_R) associated with a nonleaf node N is obtained from the intervals associated with the two children of this node. $t_L(N) = t_L$ (left child of N) and $t_R(N) = t_R$ (right child of N). For our example, the binary computation tree together with time intervals is shown in Fig. 8.

A schedule that minimizes $L_{\max}$ is now obtained by making two passes over this computation tree. The first pass is made level by level towards the root; the second is made level by level from the root to the leaves. Let P be any node in the computation tree. Let the interval associated with P be (t_L, t_R). The *set of available jobs* $A(P)$ for P consists exactly of those jobs that have a release time r_i such that $t_L \le r_i < t_R$. This set of jobs may be partitioned into two subsets, respectively, called the *used set* and the *transferred set*. The set of used jobs consists exactly of those available jobs that will be scheduled between t_L and t_R for the $L_{\max}$ problem defined by the job set $A(P)$. The remaining jobs in $A(P)$ make up the transferred set. For our example, the set of available jobs for the node representing the interval (2, 6) is {5, 8, 6, 2, 3, 1, 4, 9}. If Horn's algorithm is used on this set of jobs, then jobs 5, 8, 6, and 2 will get scheduled in the interval from 2 to 6. Hence, the used set is {5, 8, 6, 2} and the transferred set is {3, 1, 4, 9}.

In the first of the two passes mentioned above, the used and transferred sets for each of the nodes in the computation tree are determined. For a leaf node the used and transferred sets are determined by directly using Jackson's rule. If P is a leaf node for the interval (t_L, t_R), then the used set is obtained by selecting jobs from the available job set $A(P)$ for P in nondecreasing order of due times. Since jobs with the same release time have already been sorted by due times, the used set consists of the first $\min\{|A(P)|, t_R - t_L\}$ jobs in $A(P)$. The remaining jobs form the transferred set. For our example, for the interval (2, 5), the set of used jobs is {5, 8, 6} while the set of transferred jobs is {2, 3}; for the interval (5, 6), the used set is {1} and the transferred set is {4, 9}, etc. Fig. 9 shows the used and transferred sets for each of the leaf nodes for our example. The solid vertical line separates the used jobs from the transferred jobs.

For a nonleaf node, the used and transferred sets may be computed from the used and transferred sets of its children. Let P be a nonleaf node and let U_L, U_R, T_L, and T_R be the used and transferred sets for its left and right children respectively. Let (t_L, t_R), (t_L^1, t_R^1), and (t_L^2, t_R^2) be the intervals, respectively, associated with node P, its left child, and its right child. Clearly, $t_L = t_L^1$; $t_R = t_R^2$; and $t_R^1 = t_L^2$. It should be clear that if Horn's algorithm is used to schedule the available jobs $A(P)$, then the jobs in U_L will be the ones scheduled from t_L to t_R^1. The set of jobs scheduled from t_R^1 to t_R will be some subset of $T_L \cup U_R$. Let Q denote the $\min\{|T_L \cup U_R|, t_R - t_R^1\}$ jobs of $T_L \cup U_R$ that have least due times. It is not too difficult to see that Q is the subset of $A(P)$ that is scheduled by Horn's algorithm in the interval t_R^1 to t_R. Hence, the used set for P is $U_L \cup Q$ and the transferred set is $A(P) - U_L - Q$. Observe that if U_L, U_R, T_L, and T_R are in nondecreasing order of deadlines, then the set Q can be obtained by merging together U_R and T_L and selecting the first $\min\{|T_L \cup U_R|, t_R - t_R^1\}$ jobs from the merged list. Q can next be merged with U_L to obtain the used set in nondecreasing order of due times. Another merge yields the transferred set in nondecreasing order of due times. Fig. 9 gives the used and transferred sets in nondecreasing order of due times for all nodes in our example computation tree.

In the second pass, the used sets are updated so that the used set

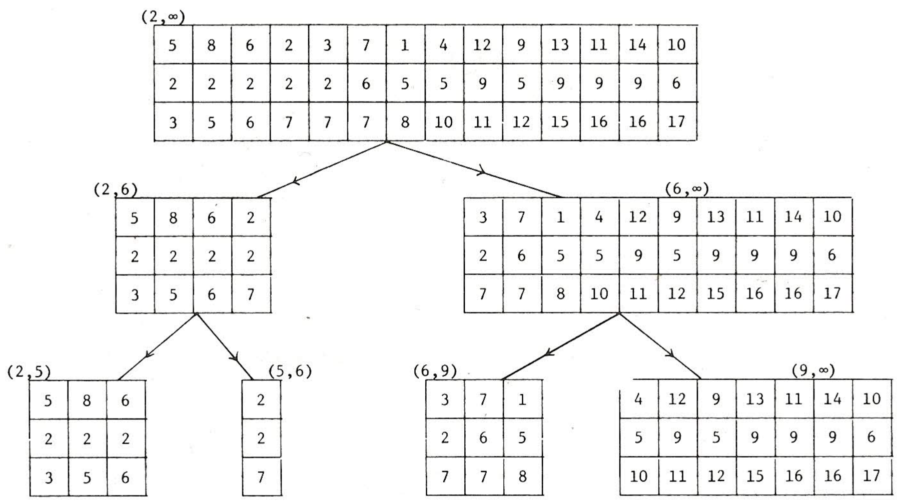


Fig. 10. Results of second pass.

for a node representing the interval (t_L, t_R) is precisely the subset of jobs (from among all n jobs) that is scheduled in this interval by Horn's algorithm when solving the $L_{\max}$ problem for the entire job set. This is done by working down the computation tree level by level starting with the root. The used set for the root node is unchanged in this pass. If P is a node whose used set has been updated, then the used sets for the left child and the right child of P are obtained in the following way. Let the interval associated with P be (t_L, t_R) and let the interval associated with its left child be (t_L, t_R^1). Let V be the subset of the used set of P consisting solely of jobs with a release time less than t_L. V may be obtained from the used set of P by first selecting the elements of P that are to be in V and then compacting these elements without changing their relative order [29]. Let U be the current used set (i.e., the one computed in the first pass) for the left child of P. Let W be the set obtained by merging U and V (note that U and V are disjoint and that both are ordered by due times). The new used set, U^1, for the left child of P consists of the first min $\{|W|, t_R^1 - t_L\}$ jobs in W. The used set for the right child of P consists of all jobs in the used set for P that are not included in U^1.

Let us now go through this second pass on our example. Let P be the root node. $(t_L, t_R) = (2, \infty)$ and $V = \phi$. Hence, the new used set for the left child of P is simply its old used set. The used set for the right child of P becomes $\{3, 7, 1, 4, 12, 9, 13, 11, 14, 10\}$. Now, let P be the right child of the root. $(t_L, t_R) = (6, \infty)$; $V = \{3, 1, 4, 9\}$; $W = \{3, 7, 1, 4, 9, 10\}$. The new used set for the left child of P is $\{3, 7, 1\}$. The new used set for the right child of P is $\{4, 12, 9, 13, 11, 14, 10\}$. Fig. 10 shows the new used sets for all the nodes in the computation tree.

From the definition of an updated used set, it follows that the schedule defined by the leaf nodes (for our example, this is job 5 at time 2, job 8 at time 3, job 6 at time 4, job 2 at time 5, etc.) minimizes $L_{\max}$. The correctness of the node updating procedure is easily seen. If P is the root node, then it represents the interval (R_1, ∞). All jobs are necessarily scheduled in this interval by Horn's algorithm. Hence, the updated used set for this node consists of all n jobs. Now, let P be any nonleaf node for which we have obtained the updated used set. Assume that this is in fact the correct updated used set, i.e., it consists exactly of those jobs scheduled by Horn's algorithm in that interval. We shall show that the updating procedure gives the correct used sets for the left and right child of P. Let t_L, t_R^1, t_R, V, W, U, and U^1 be as defined in the updating procedure. Let X be the used set for P. From the way the first pass works, it follows that only jobs from $W = U \cup V$ are candidates for scheduling by Horn's algorithm, in the interval (t_L, t_R^1). It is a simple matter to see that only min $\{|W|, t_R^1 - t_L\}$ of these can be scheduled in this interval; further these jobs are selected in nondecreasing order of due times. Hence, U^1 is correctly computed. From this it follows that the used set for the right child must be $X - U^1$.

Having established the correctness of our parallel procedure, we are ready to determine its complexity as well as the required number of PE's. The first step consists of sorting the jobs. This can be done in $0(\log^2 n)$ time using $\lfloor n/2 \rfloor$ PE's [4]. In both the first and second passes over the computation tree we are essentially performing a fixed number of merges of ordered sets at each node. Using Batcher's bitonic merge scheme ([4], [18]), a p element ordered set can be merged with a q element ordered set using $\lfloor (p + q)/2 \rfloor$ PE's in $0(\log(p + q))$ time. Hence, the overall complexity of our parallel $L_{\max}$ algorithm is $0(\log^2 n)$. The number of PE's used is $\lfloor n/2 \rfloor$. The EPU of this algorithm is $\Omega(n \log n/(n/2 \log^2 n)) = \Omega(1/\log n)$.

Our parallel $L_{\max}$ algorithm for the case $m = 1$ easily generalizes to the case $m \geq 1$. The two passes over the computation tree are changed so that all uses of $t_R - t_L$ and $t_R^1 - t_L$ are replaced by $m(t_R - t_L)$ and $m(t_R^1 - t_L)$, respectively. The schedule is obtained from the updated used sets of the leaf nodes. The ith job in this used set is assigned to the i mod m + 1th machine.

2) $m = 1$ and Preemptions Permitted: Horn's [14] algorithm for this problem is quite similar to the sequential algorithm for the case discussed in Section III-A1) and also has a sequential complexity that is $0(n \log n)$. A schedule with minimum $L_{\max}$ is obtained by starting at the first release time and considering an available job i with least due time. Let the processing time of this job be p. Let the time to the next release time be t and let the current time be T. Job i is scheduled from T to T + min $\{p, t\}$. The current time changes from T to T + min $\{p, t\}$ and the remaining processing time for job i becomes p − min $\{p, t\}$. Next, from the available job set at the current time T a job with minimum due time is selected for processing, and so on.

The parallel algorithm of Section III-A1) can be adapted to this case. Jobs are sorted as before and two passes are made over the tree. In the first pass, used and transferred sets are computed for each node. In the second pass, the used sets are updated. For the first pass, the used and transferred sets for the leaf nodes are obtained by computing the partial sum sequence for the ordered set of available jobs for each leaf (see the algorithm of Fig. 6). Next, for each leaf we determine the first partial sum j (if any) that exceeds the value of $t_R - t_L$ for that node. If there is no such partial sum, then all the available jobs are used. If there is, then the used set consists of jobs $1, 2, \cdots, j - 1$ together with a fraction f of job j. This fraction is chosen such that the sum of the processing times of jobs $1, 2, \cdots, j - 1$ and j times that of job j equals $t_R - t_L$. The transferred set consists of $1 - f$ of job j together with the remaining jobs.

For the nonleaf nodes, the used and transferred sets are computed from the corresponding sets for the left and right children. Let P be a nonleaf node. Let Q and S be its left and right children respectively. The used set for P is obtained by merging (according to due times) the transferred set of Q with the used set of S, to obtain W. The partial sums for W are computed and W is partitioned into $W1$ and $W2$ such that the sum of the proccessing times for the jobs in $W1$ equals min {sum of processing times in W, $t_R^2 - t_L^2$} where (t_L^2, t_R^2) is the interval associated with node S. Observe that this partitioning of W may require us to split one of the jobs in W in the same way as was done for leaf nodes. The used set for P is obtained by merging together $W1$ and the used set for Q. The transferred set for P is obtained by merging together $W2$ and the transferred set for S.

The updating of the second pass is also carried out in a manner similar to that used in Section III-A1). The updated used set for the root node consists of all n jobs. Let P be a node for which the updated used set has been computed. Let (t_L, t_R) be the interval associated with P. Let Q and S, respectively, be the left and right children of P. Let the interval associated with Q be (t_L, t_R^1). Define V to be the set of all jobs in the used set of P that have a release time less than t_L. Merge V and the current used set of Q together. Let the resulting ordered set be W. Compute the partial sums for W and partition W into $W1$ and $W2$ as was done in the first pass. Once again, it may be necessary to split a job into two to accomplish this. The used set for Q is $W1$. The remaining jobs in the used set of P (including possibly a remaining fraction of a job that went into $W1$) constitute the used set for S.

Once the updated used sets for the leaves have been computed, a schedule minimizing $L_{\max}$ is obtained by scheduling the used sets of the leaves in the intervals associated with them. For each such interval, the scheduling is in nondecreasing order of due time.

The correctness of the algorithm described above follows from the correctness of Horn's algorithm and the discussion in Section III-A1). The algorithm can be run in $0(\log^2 n)$ time using at most $3n/2$ PE's. Note that because jobs may split, we may at some level have a total of $n + 2k$ jobs (or job parts). Recall that k denotes the number of distinct release times and that at each node at most one additional job split can occur. Because of the effective increase in number of jobs, more than $\lfloor n/2 \rfloor$ PE's are needed here, while only $\lfloor n/2 \rfloor$ were needed in Section III-A1). The EPU is still $\Omega(1/\log n)$.

3) *Precedence Constraints:* Suppose that the set of jobs to be scheduled defines a partial order $<$. $i < j$ means that the processing of job j cannot commence until the processing of job i has been completed. Let (r_i, d_i, p_i) be the release, due, and processing times of job i. Modify the release and due times as below.

$$r_i' = \max\left\{r_i, \max_{j<i}\{r_j' + p_j\}\right\}$$

$$d_i' = \min\left\{d_i, \min_{i<j}\{d_j' - p_j\}\right\}$$

Rinooy Kan [31] has observed that a schedule minimizing $L_{\max}$ when $p_i = 1$, $m = 1$, the r_i's are integer, and $<$ is a partial order can be obtained by simply using Horn's algorithm [cf. Section III-A1)] on the jobs $(r_i', p_i = 1, d_i')$, $1 \le i \le n$ with no precedence constraints. Since the modified release and due times can be computed in $0(\log^2 n)$ time using the critical path algorithm of [9], a schedule minimizing $L_{\max}$ in the presence of precedence constraints can be obtained in $0(\log^2 n)$ time ($m = 1$, $p_i = 1$). The number of PE's needed by the algorithm of [9] is $n^2/\log n$, so the EPU of the resulting algorithm is $\Omega(n^2 \log n/(n^2 \log^2 n))) = \Omega(1/(n \log n))$.

When $m = 1$, a partial order $<$ is specified, and preemptions are allowed, a schedule minimizing $L_{\max}$ can be obtained by computing modified release and due times as above and then using the algorithm of Section III-A2) on the modified jobs. The resulting algorithm has complexity $0(\log^2 n)$, uses $0(n^2/\log n)$ PE's, and has an EPU that is

$$\Omega\left(\frac{1}{n \log n}\right).$$

B. *Minimizing Total Costs*

Let (r_i, d_i, p_i, w_i), $1 \le i \le n$ define n jobs. Let S be any machine schedule for these jobs. the completion time c_i of job i is the time at which it completes processing. Job i is tardy iff $c_i > d_i$. The tardiness T_i of job i is $\max\{0, c_i - d_i\}$. When $p_i = 1$, Horns [14] algorithm described in Section III-A1) also finds a schedule that minimizes ΣT_i, ΣL_i, and $T_{\max}$ [31, p. 80].

A schedule that minimizes $\Sigma w_i c_i$ when $p_i = 1$, $1 \le i \le n$ and $m = 1$ can be obtained by extending Smith's rule (see Rinnooy Kan [31]). Smith's rule [35] minimizes $\Sigma w_i c_i$ when $r_i = 0$, $1 \le i \le n$. It essentially schedules jobs in nondecreasing order of p_i/w_i. The extension to the case when $p_i = 1$, $1 \le i \le n$ and the r_is may be different (but integer) works in the following way. Scheduling is done time slot by time slot. From the set of available jobs for any slot, a job with least $1/w_i$ (or equivalently, maximum w_i) is selected and scheduled in this slot. This procedure is quite similar to that used for the $L_{\max}$ problem with $p_i = 1$ [see Section III-A1)]. The only difference is that Smith's rule replaces the use of Jackson's rule. The used and transferred sets are now kept in nonincreasing order of weights. Note that this method is easily extended to the case $m > 1$.

When Σc_i is to minimized $m = 1$ and preemptions are permitted, the algorithm of Section III-A2) can still be used. This time, however, the used and transferred sets are maintained in nondecreasing order of p_i rather than d_i [31].

Number of Tardy Jobs

Now, let us consider the problem of minimizing the number of tardy jobs when $m = 1$ and all jobs have the same release time. Without loss of generality, we may assume that all jobs have a release time $r_i = 0$. The fastest sequential algorithm for this problem is due to Hodgson and Moore [23]. It consist of the following three steps:

Step 1: Sort the n jobs into nondecreasing order of due times. Initialize the set R of tardy jobs to be empty.

Step 2: If there is no tardy job in the current sorted sequence, then append the jobs in R to this sequence. This yields the desired schedule. Stop.

Step 3: Find the first tardy job in the current sorted sequence. Let this be in position J. Find the job with the largest processing time from amongst the first j jobs in this sequence. Remove this job from the sequence and add it to R. Go to step 2.

The time complexity of the Hodgson and Moore algorithm is $0(n \log n)$. As in the case of the Hodgson and Moore algorithm, our parallel algorithm for this problem begins by sorting the jobs into nondecreasing order of due times. Within due times, jobs are sorted by p_i. Let $D_1, D_2, \cdots$, and $D_k (D_1 < D_2 < \cdots < D_k)$ be the k distinct due times associated with the n jobs. Let $D_0 = 0$. We next consider the unique complete binary tree that has exactly k leaves. If the leaf nodes of this tree are considered from left to right, then with the ith leaf we associate the interval (D_{i-1}, D_i). The interval associated with a nonleaf node is (t_1, t_2) iff there exists t_3 such that (t_1, t_3) and (t_3, t_2) are the intervals, respectively, associated with its left and right children. If the interval (t_1, t_2) is associated with some node P, then all jobs with a due time d such that $t_1 < d \le t_2$ are associated with that node.

The set $J(P)$ of jobs associated with any node P may be partitioned into two sets $S(P)$ and $R(P)$. $S(P)$ and $R(P)$ are defined in the following way. Consider the problem of obtaining a schedule that minimizes the number of tardy jobs for $J(P)$ assuming that all jobs in $J(P)$ have a release time t_1[(t_1, t_2) is the interval associated with P]. $S(P)$ is the set of nontardy jobs in this schedule while $R(P)$ is the set of tardy jobs. It is well known [16] that if all jobs in $S(P)$ are scheduled in nondecreasing order of due times, then no job in $S(P)$ will be tardy. From the definition of S and R, it is clear that S(root) defines the set of nontardy jobs in a schedule for all n jobs that minimizes the number of tardy jobs. These jobs may be scheduled at the front of the schedule in nondecreasing order of due times. The remaining jobs can be scheduled, in any order, after the jobs in S(root).

For a leaf node P, $S(P)$ and $R(P)$ are easily computed. First the partial sum sequence for $J(P)$ is obtained (recall that the jobs associated with P are in nondecreasing order of p_i). Let the interval associated with P be (t_1, t_2). All jobs with a partial sum that is less than or equal to $t_2 - t_1$ are in $S(P)$. The remainder are in $R(P)$.

Let us consider an example. Fig. 11(a) shows a set of 10 jobs. In

i	1	2	3	4	5	6	7	8	9	10
p_i	4	3	5	6	4	3	3	4	3	3
d_i	15	25	8	8	15	25	17	25	8	25

(a)

i	9	3	4	1	5	7	2	6	10	8
p_i	3	5	6	4	4	3	3	3	3	4
d_i	8	8	8	15	15	17	25	25	25	25

(b)

Fig. 11.

Fig. 11(b), these jobs have been ordered by due times and within due times by p_i. There are four distinct due times, and we have $D(0{:}4) = (0, 8, 15, 17, 25)$. Fig. 12 shows the complete binary tree and four leaves. The interval associated with each node is also given. The S and R sets for each of the leaf nodes are also shown.

The computation of S and R for a nonleaf node P is done using the S and R sets of its left child Q and its right child T. Let the interval associated with Q and T, respectively, be (t_L, t_R^1) and (t_R^1, t_R). It is clear that $S(T) \subset S(P)$ and that $R(Q) \subset R(P)$. To get the remaining jobs in $S(P)$, we merge together the jobs in $S(Q)$ and $R(T)$. Let the resulting ordered set be W. The partial sum sequence of the processing times of the jobs in W is next computed. Let V be the subset of W consisting of jobs that have a partial sum sequence no more than $t_R^1 - t_L$. Let $X = W - V$. Clearly, $V \subset S(P)$. However, $V \cup S(T)$ may not equal $S(P)$ as it is possible for (at most) one of the jobs in X to also be in $S(P)$. To determine this job, we first determine for each due time D_i, $t_R^1 < D_i \le t_R$, a job in X that has least processing time amongst all jobs in x with due time D_i. If there are no jobs in X with a certain due time D_i, then no job corresponding to this due time is selected. Let the set of jobs determined in this way be $U = \{J_1, J_2, \cdots, J_q\}$. Let $\Delta = t_R^1 - t_L - \sum_{i \in V} p_i$. For each due time $D_i, t_R^1 < D_i \le t_R$, determine the sum of the processing times of all jobs in $S(T)$ with due times no more than D_i. Let this sum be Y_i. Let $\Delta_i = D_i - Y_i - t_R^1$. Now, compute $\gamma_i = \min_{j \ge i} \{\Delta_j\}$. It can be seen that the job (if any) in U with due time D_i can be in $S(P)$ only if its processing time is less than or equal to $\Delta + \gamma_i$. This information is used to remove from U those jobs that cannot possibly be in $S(P)$. From the remaining jobs, the job r with minimum processing time is selected and added to $S(P)$. $R(P) = R(Q) \cup (X - \{r\})$. The S and R sets for all nonleaf nodes in our example are specified in Fig. 12.

The sets U and $\{\Delta_i\}$ can be computed in $0(\log n)$ time using $0(n)$ PE's if S and R are available in nondecreasing order of due times (so it is necessary to keep two copies of each S and R; one ordered by processing times and one by due times). The γ_is may be computed in $0(\log n)$ time using $0(n/\log n)$ PE's using a modified version of the partial sums algorithm. Merging $S(Q)$ and $R(T)$ by processing times or by due times requires $0(\log n)$ time and $n/2$ PE's. So, all the work needed to be done at any level can be accomplished in $0(\log n)$ time with $0(n)$ PE's. The overall complexity of our parallel algorithm is therefore $0(\log^2 n)$ and its EPU is $\Omega(1/\log n)$.

Job Sequencing with Deadlines

The problem of minimizing the sum of the weights of the tardy jobs is commonly referred to as the job sequencing with deadlines problem [15]. It is assumed that $r_i = 0$, and $p_i = 1, 1 \le i \le n$. When the assumption $p_i = 1$ is not made, the problem is known to be NP-hard [17]. We shall now proceed to show how the binary tree method leads to an efficient parallel algorithm for this problem. We shall explicitly consider only the case $m = 1$. When $m > 1$, the problem can be transformed into an equivalent $m = 1$ problem. Further, all the d_i's are assumed to be integers.

An $0(n \log n)$ sequential algorithm for this problem appears in [15].

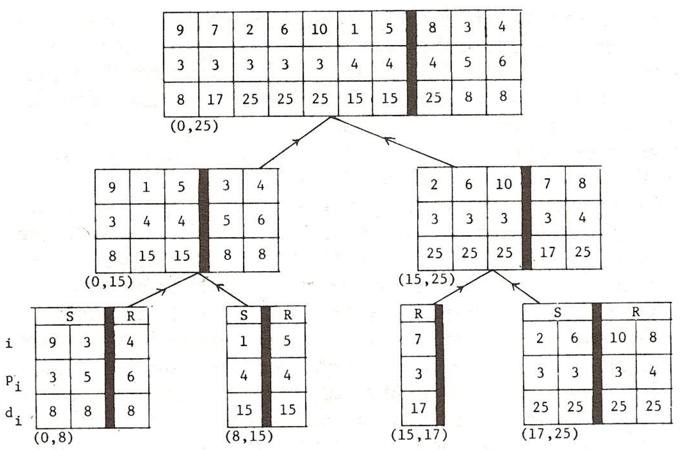


Fig. 12.

This algorithm builds an optimal schedule by first determining the set of jobs that are to be completed by their due times. This is done by considering the jobs in nonincreasing order of weights. The job currently being considered is added to the set of selected jobs iff it is possible to schedule this job and all previously selected jobs in such a way that all of them complete by their respective due times.

In our parallel algorithm, we begin by sorting the jobs by due times. Jobs with the same due time are sorted into nonincreasing order of weight. Let the distinct due times by $D_1, D_2, \cdots, D_k$ $(D_1 < D_2 < \cdots < D_k)$. Let $D_0 = 0$. The computation tree to use is the unique complete binary tree with k leaves. Consider these leaves left to right. With leaf i, we associate the interval (D_{i-1}, D_i), $1 \le i \le k$. Let P be a nonleaf node. Let the intervals associated with its left and right children, respectively be (t_L, t_R^1) and (t_R^1, t_R). The interval associated with P is (t_L, t_R). The interval associated with the root is therefore $(0, D_k)$.

The set $J(P)$ of jobs associated with node P consists precisely of those jobs that have a due time d_i such that $t_L < d_i \le t_R$ where (t_L, t_R) is the interval associated with P. With each node P, we may also associate two sets of jobs, $S(P)$ and $R(P)$. Consider the job sequencing with deadlines problem defined by the job set $J(P)$. Assume that all jobs have a release time t_L. $S(P)$ consists exactly of those jobs in $J(P)$ that will be scheduled to finish by their due times in an optimal schedule for $J(P)$. $R(P)$ consists of the remaining jobs in $J(P)$. Once S(root node) is known, the optimal schedule for the overall job sequencing problem is also known.

For the leaf nodes, $S(P)$ and $R(P)$ are easily obtained. For each leaf node P, $S(P)$ consists of the $t_R - t_L$ jobs of $J(P)$ with largest weight. If P is a nonleaf node, $S(P)$ and $R(P)$ are computed from the S and R sets of its children. Let Q and T, respectively, be the left and right children of P. Let the intervals associated with Q and T, respectively, be (t_L, t_R^1) and (t_R^1, t_R). Let $W = S(Q) \cup R(T)$ and let V be the set consisting of the $\min\{|W|, t_R^1 - t_L\}$ jobs of W with largest weights. It is not too difficult to see that $S(P) = V \cup S(T)$. Hence, $R(P) = J(P) - S(P) = R(Q) \cup (W - S(P))$.

Once the S and R sets have been computed, the optimal schedule can be obtained by sorting S(root) by due times and appending the jobs in R(root) to the end.

Since the S and R sets are maintained in nonincreasing order of weights, the merging required at each node to compute S and R can be carried out using a parallel bitonic merge. Hence, all the computation needed at each level of the computation tree can be performed in $0(\log n)$ tiome using $n/2$ PE's. The overall complexity for our job sequencing with deadlines algorithm is $0(\log^2 n)$ and the EPU is $\Omega(1/\log n)$. (In [10] Dekel and Sahni show how to solve the job sequencing problem in $0(\log n)$ time. This algorithm does not use the binary tree method and has an EPU which is considerably inferior to that of the algorithm developed here.)

Finally, we note that the parallel algorithm developed to minimize the number of tardy jobs when $m = 1$ and $r_i = 0$, can be adapted to obtain a one machine schedule that minimizes the sum of the weights of the tardy jobs provided that all jobs have agreeable weights. (All jobs have agreeable weights iff $p_i < p_j$ implies $w_i \leq w_j$ for all i and j.) The sequential algorithm for this problem is an extension of the Hodgson–Moore algorithm to minimize the number of tardy jobs. This extension is due to Lawler [21]. Also, Sidney's [34] extension which takes into account jobs that must necessarily be completed by their due times can also be solved by a modified version of our algorithm.

IV. Conclusions

We have demonstrated that the binary computation tree is a very important tool in the design of efficient parallel algorithms. The binary tree method is closely related to the divide-and-conquer approach used to obtain many efficient sequential algorithms [15]. While divide-and-conquer algorithms do use an underlying computation structure that is a tree, the use of this tree is implicit. Further, only one pass over this tree can be made as partial results computed in the various nodes are not saved for use in further passes. In this respect, the binary tree method is more general than divide-and-conquer. The single pass algorithms discussed in this paper can, however, be just as well viewed as divide-and-conquer algorithms.

While all the parallel algorithms discussed in this paper have assumed that as many PE's as needed are available, they can be run quite easily using fewer PE's. The complexity of course will increase by a factor of q/k where k is the number of PE's available and q is the number assumed in the paper.

References

[1] T. Agerwala and B. Lint, "Communication in parallel algorithms for Boolean matrix multiplication," in *Proc. IEEE Int. Conf. Parallel Process.*, 1978, pp. 116–153.

[2] F. Arjomandi, "A study of parallelism in graph theory," Ph.D. dissertation, Dep. Comput. Sci., Univ. Toronto, Ont., Canada, Dec. 1975.

[3] K. R. Baker and Z.-S. Su, "Sequencing with due-dates and early start times to minimize maximum tardiness," *Naval Res. Logist. Quart.* vol. 21, pp. 171–176, 1974.

[4] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32. Montvale, NJ: AFIPS Press, 1968, pp. 307–314.

[5] ——, "MPP—A massively parallel processor," in *Proc. IEEE Int. Conf. Parallel Process.*, 1979, p. 249.

[6] J. Blazewicz, "Simple algorithm for multiprocessor scheduling to meet deadlines," *Info. Process. Lett.*, vol. 6, no. 5, pp. 162–164, 1977.

[7] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. Ass. Comput. Mach.*, vol. 21, pp. 201–206, Apr. 1974.

[8] L. Csanky, "Fast parallel matrix inversion algorithms," in *Proc. 6th IEEE Symp. Found. Comput. Sci.*, Oct. 1975, pp. 11–12.

[9] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," *SICOMP*, vol. 10, pp. 657–675, 1981.

[10] E. Dekel and S. Sahni, "Parallel scheduling algorithms," *Oper. Res*, vol. 31, pp. 24–49, 1983.

[11] D. Eckstein, "Parallel graph processing using depth-first search and breadth first search," Ph.D. dissertation, Univ. Iowa, Iowa City, IA, 1977.

[12] D. S. Hirschberg, "Parallel algorithms for the transitive closure and the connected component problems," in *Proc. 8th Ass. Comput. Mach. Symp. Theor. Comput.*, May 1976, pp. 55–57.

[13] ——, "Fast parallel sorting algorithms," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 657–661, Aug. 1978.

[14] W. A. Horn, "Some simple scheduling algorithms," *Naval Res. Logist. Quart.*, vol. 21, pp. 177–185, 1974.

[15] E. Horowitz, and S. Sahni, *Fundamentals of Computer Algorithms.* Potomac, MD: Computer Science Press, 1978.

[16] J. K. Jackson, "Scheduling a production line to minimize tardiness," Management Science Research Project, Univ. California, Los Angeles, CA, Res. Rep. 43, 1955.

[17] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations.* R. E. Miller, and J. W. Thatcher, Eds. New York: Plenum, 1972.

[18] D. E. Knuth, *The Art of Computer Programming Vol. 3: Sorting and Searching.* Reading, MA: Addison-Wesley, 1973.

[19] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Trans. Comput.*, vol. C-25, pp. 496–503, May 1976.

[20] T. Lang and H. Stone, "A shuffle exchange network with simplified control," *IEEE Trans. Comput.*, vol. C-25, pp. 55–65, Jan. 1976.

[21] E. L. Lawler, "Sequencing to minimize the weighted number of tardy jobs," *Rev. Francaise Automat. Informat. Recherche Operationalle*, vol. 10.5, suppl., pp. 27–33, 1976.

[22] J. K. Lenstra, "Sequencing by enumerative methods," in *Mathematical Centre Tract 69.* Amsterdam, The Netherlands: Mathematisch Centrum, 1977.

[23] J. M. Moore, "An *n* job, one machine sequencing algorithm for minimizing the number of late jobs," *Management Sci.*, vol. 15, pp. 102–109, 1968.

[24] D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and for switching," *J. Ass. Comput. Mach.*, vol. 22, pp. 195–201, Apr. 1975.

[25] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," *JCSS*, vol. 7, pp. 189–198, 1973.

[26] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol. C-28, pp. 2–7, Jan. 1979.

[27] ——, "An optimal routing algorithm for mesh connected parallel computer," *J. Ass. Comput. Mach.*, vol. 27 no. 1, pp. 6–29, 1980.

[28] ——, "Parallel permutation and sorting algorithms and a new generalized connection network," *J. Ass. Comput. Mach.*, to be published.

[29] ——, "Data broadcasting in SIMD Computers," *IEEE Trans. Comput.*, vol. C-30, pp. 101–107, Feb. 1981.

[30] F. P. Preparata, "New parallel-sorting schemes," *IEEE Trans. Comput.*, vol. C-27, pp. 669–673, July 1978.

[31] A. H. G. Rinooy Kan, "Machine scheduling problems: Classification, complexity, and computations," Nighoff, The Hague, The Netherlands, 1976.

[32] C. Savage, "Parallel algorithms for graph theoretic problems," Ph.D. dissertation, Univ. Illinois, Urbana, IL, Aug. 1978.

[33] H. Siegal, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.*, vol. C-28, pp. 907–917, 1979.

[34] J. B. Sidney, "An extension of Moore's due date algorithm," in *Symp. Theory Scheduling and its applications, Lecture Notes in Economics and Mathematical Systems, 86.* S. E. Elmagharaby, Ed. Berlin, Germany: Springer, 1973, pp. 393–398.

[35] W. E. Smith, "Various optimizers for single-stage production," *Naval Res. Logist. Quart.*, vol. 3, pp. 59–66, 1956.

[36] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, 1971.

[37] C. D. Thompson, "Generalized connection networks for parallel processor interconnection," *IEEE Trans. Comput.*, vol. C-27, pp. 1119–1125, Dec. 1978.

[38] C. D. Thompson and H. T. Kung, "Sorting on a mesh connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, no. 4, pp. 263–271, 1977.