*PARALLEL ALGORITHMS & ARCHITECTURES*
*M. Cosnard et al. (editors)*
*© Elsevier Science Publishers B.V. (North-Holland), 1986*

# A PARALLEL SCHEDULING ALGORITHM FOR MINIMIZING THE NUMBER OF UNSCHEDULED JOBS

Günter ROTE

Institut für Mathematik, Technische Universität Graz
Kopernikusgasse 24, A-8010 Graz, Austria.

We consider the problem of scheduling n jobs with different processing times on one machine subject to a common release date and different due-dates, in order to maximize the number of jobs that are finished in time.

The most efficient sequential algorithm which is known for this problem is due to Moore and Hodgson and runs in $O(n \log n)$ time. We present a parallelization of this algorithm which runs in $O(\log^2 n)$ time on $n^2$ processors. Our model of computation is the single instruction stream, multiple data stream (SIMD) shared memory model without write conflicts but with read conflicts.

Our approach is based on the binary tree method of Dekel and Sahni. It requires a thorough analysis of the behavior of the sequential algorithm. We show that a parallel algorithm for the same scheduling problem given by Dekel and Sahni relies on an erroneous assumption.

## 1. INTRODUCTION

Parallel algorithms for scheduling problems have been considered in a series of papers by Dekel and Sahni [1981, 1983a, 1983b, 1984].
In this paper, we consider the problem of scheduling n jobs with different processing times on one machine subject to a common release date and different due-dates, in order to maximize the number of jobs that are scheduled in time. This problem has been dealt with in Dekel and Sahni [1983a], but as will be shown below (at the end of section 2), their algorithm relies on an erroneous assumption.

An efficient sequential algorithm for this problem has been known for a long time (Moore [1968]). Our parallelization of this algorithm requires a careful analysis of its behavior. It is based on the binary tree method of Dekel and Sahni [1983a]. The model of computation is the single instruction stream, multiple data stream (SIMD) shared memory model without write conflicts but with read conflicts allowed.

In section 2, we define the problem, and we review the Hodgson-Moore algorithm for its solution. We discuss the properties of the solution and we give extensions and modifications which will allow the problem to be treated in parallel. Sections 3 describes the algorithm, section 4 contains the details of its parallel implementation, and section 5 concludes the paper.

This work was supported by the Austrian science foundation (Fonds zur Förderung der wissenschaftlichen Forschung) Project S32/01.

## 2. THE SCHEDULING PROBLEM

### 2.1. Definition of the problem

We consider the following scheduling problem:
There is one machine which is continuously available and which can process one job at a time. There is a number of jobs $i=1,\dots,n$, with a common release date r for all jobs, and a processing requirement $p_i$ and a due-date $d_i$ for each job.

We want to find the maximum number of jobs that can be scheduled such that the jobs are processed without starting earlier than the release date and without finishing after their due-dates (a **feasible** schedule).

### 2.2. The Moore-Hodgson algorithm

The following facts are known:
As subset S of the jobs can be scheduled such that no job is tardy (i. e., finishes after its due-date) if and only if the schedule produced by the EDD-rule (earliest due-date) is feasible. (The EDD-schedule is obtained by processing the jobs in order of increasing due-dates, starting from the release date.) Such a set S is called a **feasible** set of jobs. In other terms, a set of jobs with due-dates $d_1 \leqq d_2 \leqq \dots \leqq d_i$ and processing times $p_1, p_2, \dots, p_i$ is feasible if and only if

$$\text{for all } j=1,2,\dots,i: \quad r + (p_1+p_2+\dots+p_j) \leqq d_j \; .$$

The fastest sequential algorithm for the problem is due to Hodgson (Moore [1968]) and takes O(n log n) time. It is a greedy-type procedure and runs as follows:

(i) The jobs are considered in increasing order of due-dates. They are scheduled in this order, beginning at the release date.

(ii) Each job considered is appended to the list of jobs scheduled so far. If its finishing time exceeds its due-date, we find the longest job in the list of jobs scheduled so far (including the job just appended) and cancel it from the list (forever; it will never be considered again).

In any case all of the jobs now in the list finish in time. We proceed by considering the next job.

(iii) The final list obtained is the set of jobs that are scheduled in an optimal schedule.

Formally, the algorithm runs as follows:

```
Initially, we assume that $d_1 \leqq d_2 \leqq \dots \leqq d_n$.
$f := r$; $S := \emptyset$;
**for** i **from** 1 **to** n **do**
    $f := f + p_i$; $S := S \cup \{i\}$;
    **if** $f > d_i$ **then**
        $j := \text{argmax } \{p_j \mid j \in S\}$;
        $f := f - p_j$; $S := S \setminus \{j\}$;
```

### 2.3. Example:

$n=6$, $r=0$, the given $d_i$ and $p_i$ values are shown in the following table. The optimal schedule constructed by the algorithm consists of the jobs 2, 3, 5, and 6. The start times $a_i$ of these jobs in the schedule is given in the last line. Job 1 is canceled when job 3 is considered, and job 4 is canceled when it is considered itself.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $d_i$ | 8 | 9 | 10 | 11 | 16 | 17 |
| $p_i$ | 6 | 4 | 3 | 5 | 7 | 2 |
| $a_i$ | - | 0 | 4 | - | 7 | 14 |

## 2.4. Analysis of the algorithm

THEOREM 1: The set S constructed by the algorithm is optimal in two ways:

(1) It is a maximum-cardinality feasible set of jobs.

(2) Among the feasible sets with equal cardinality, it has minimum total processing time of the jobs (or equivalently, minimum finishing time).

PROOF: Assume that T is an optimal set of jobs but T contains jobs that are not in S. We shall argue that T can be transformed into a set T' which

(a) is still feasible,

(b) contains as many jobs as before,

(c) has total processing time no longer than before,

(d) is therefore optimal as well, and

(e) has one more job in common with S.

Let j by the first job in T\S (with smallest due-date). We denote by $S_0$ the set of jobs from which the job j was canceled in the algorithm. No feasible set of jobs can contain all jobs in $S_0$. Therefore, there must be some job k of $S_0$ not contained in T. By the construction of j, $p_k \le p_j$. We construct the set T' by replacing j by k in T. If the EDD-schedule for the set T was feasible, then the EDD-schedule for T' is also feasible (see figure 1) since

(i) the jobs in $S_0$, which are scheduled first, are finished in time because they are a subset of $S_0 \setminus \{j\}$, which is a feasible set;

(ii) the remaining jobs, which are scheduled afterwards, are finished by $p_j - p_k$ earlier than in T, and therefore in time.

The remaining desired properties of T', (b), (c), (d), and (e), are trivial.

By repeating the above transformation as many times as necessary, we can transform T into an optimal set which is a subset of the feasible set S and which is therefore equal to S. ∎

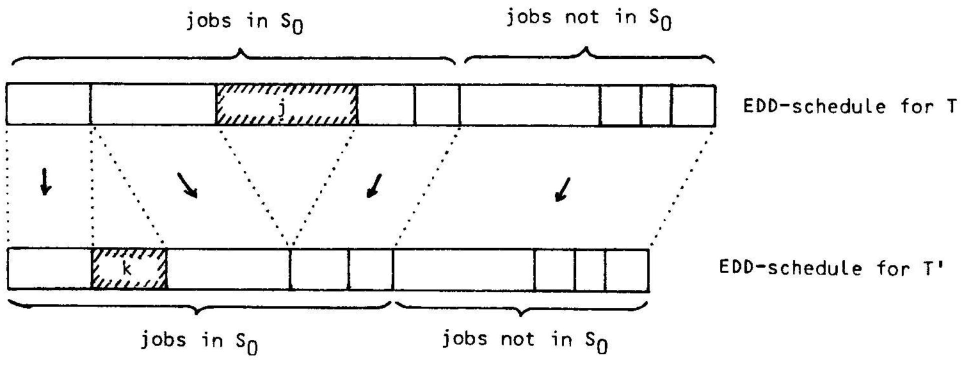


FIGURE 1
Construction of the EDD-schedule T' from the schedule T.

## 2.5. Refinement of the algorithm

One undesirable property of the algorithm, which is partly responsible for the inherently sequential character of the algorithm, is that when job j is considered, the algorithm does not decide conclusively whether this job will be in the final schedule.

We shall show that, with some preprocessing, a final decision whether each job belongs to the optimal schedule or not, can be made in a sequential manner.

Assume that the jobs are initially ordered by their due-dates, with ties broken arbitrarily, and that, when selecting the job with largest processing time in a set of jobs, ties are broken in favor of the job with the smallest number.

If a job i is put into the list at some time, then it will not finally remain in the list unless all subsequent jobs with shorter (or equal) processing time are also in the final list, because when the algorithm subsequently looks for a job to be removed, it will rather select i than any of the subsequent shorter jobs. Now let's assume that we know which of the jobs before i belong to the optimal solution, and these jobs constitute the current list. When we consider job i in the algorithm, we could test whether the schedule formed by appending i and all subsequent shorter jobs to the current list is feasible. If it is not, we can reject i right away, as we have seen above. If it is, then we can accept i to the final solution, since subsequent jobs with larger processing time can do no harm because they would rather be canceled themselves before job i is canceled, and the remaining jobs can be scheduled feasibly.

Thus we can set up the following variation of the algorithm:

We assume that $d_1 \leq d_2 \leq \ldots \leq d_n$.

```
A)  Preprocessing:
    for each i=1,...,n do
          $s_i := \min \{ d_j - \sum\{p_k \mid i \leq k \leq j, p_k \leq p_i\} \mid i \leq j \leq n \}$ .
          $s_i$ is the latest possibly start time for job i if all jobs j with $p_j \leq p_i$
          are to be scheduled in time. (Actually, the minimum need only be taken
          over all j with $p_j \leq p_i$.)

B)  Main loop:
    $f_0 := r$; $S := \emptyset$;
    for i from 1 to n do
          if $f_{i-1} \leq s_i$ then $f_i := f_{i-1} + p_i$; $S := S \cup \{i\}$;
                                  (Job i will be scheduled.)
                    else $f_i := f_{i-1}$;
                                  (Job i will not be scheduled.)
```

The subscripts of the variable f have been added to distinguish between the values in different steps of the algorithm.

2.6. Continuation of example 2.3

The following table shows the $p_i$ and $d_i$ values as given in section 2.3, the computed $s_i$ values and the values of $f_i$ during the course of the algorithm. The jobs of the optimal schedule constructed are indicated by a star.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $d_i$ | | 8 | 9 | 10 | 11 | 16 | 17 |
| $p_i$ | | 6 | 4 | 3 | 5 | 7 | 2 |
| $s_i$ | | -7 | 3 | 7 | 6 | 8 | 15 |
| $f_i$ | 0 | 0 | *4 | *7 | 7 | *14 | *16 |

2.7. Example:

n=4, r=-2, the given $d_i$ and $p_i$ values and the computed $s_i$ values are shown in the following table, and an optimal schedule is shown in figure 2.

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $p_i$ | 2 | 3 | 4 | 5 |
| $d_i$ | 3 | 5 | 8 | 10 |
| $s_i$ | 1 | 2 | 4 | 5 |

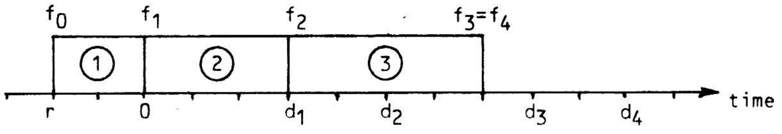


FIGURE 2
Optimal schedule for example 2.7.

## 2.8. Towards a parallelization

The preprocessing part is parallelizable in a straightforward way, but the main loop looks inherently sequential. Nothing can be said about $f_i$ before $f_{i-1}$ is known, except for the abstract functional relationship which the algorithm establishes between $f_{i-1}$ and $f_i$, or, more generally, between $f_i$ and $f_j$ for $i<j$. We denote this function by $F_{ij}$. The function $F_{ij}$ gives the value of $f_j$ for all possible values of $f_i$, and thus it can be dealt with without having a specific value of $f_i$. This is the key to decomposing the problem of computing $f_n$ starting from $f_0$ into independent parts which can be executed in parallel. The price that one has to pay is that one has to compute a representation of $F_{ij}$ for the whole domain of values of $f_i$, and not just for a single value.

It is clear that the composition of $F_{ij}$ and $F_{jk}$ is $F_{ik}$ and $F_{0n}(f_0)=F_{0n}(r)=f_n$ is the finishing time of the optimal schedule. Figure 3 shows the functions $F_{0i}$, $i=1,2,3,4$ for the example of section 2.7. The corresponding set S of jobs in the optimal solution is also given for all values of $f_0=r$. $F_{0i}$ can be obtained inductively from $F_{0,i-1}$ by applying $F_{i-1,i}$. $F_{i-1,i}$ is the function:

if $f_{i-1}\leqq s_i$ then $F_{i-1,i}(f_{i-1}) = f_{i-1}+p_i$ else $F_{i-1,i}(f_{i-1}) = f_{i-1}$;

i. e., the graph of $F_{0i}$ is obtained from the graph of $F_{0,i-1}$ by shifting everything which is below the horizontal line at $f_{i-1}=s_i$ upwards by $p_i$.

THEOREM 2: Let $0\leqq i<j\leqq n$ and $j-i=m$. Then the function $F_{ij}$ has the following properties:

(1) It is piecewise linear with constant slope equal to 1.

(2) The linear pieces are defined on half-open intervals which are open on the left side and closed on the right side.

(3) There are at most $\binom{m+1}{2}$ discontinuities ("breakpoints").

(4) At the discontinuities, there are in total at most m different ordi-

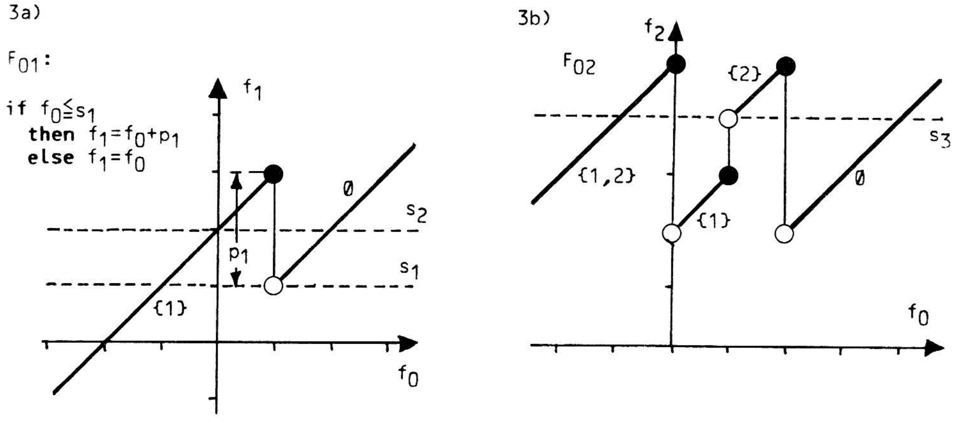
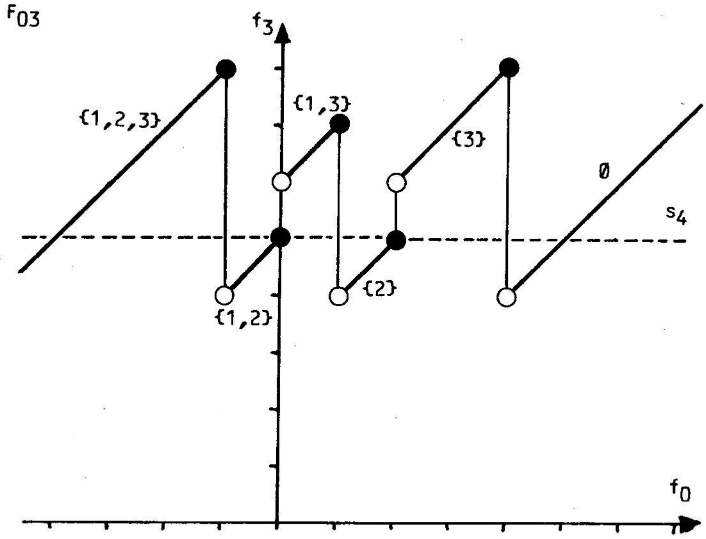


FIGURE 3
The functions $F_{01}$ and $F_{02}$.

3c)



3d)

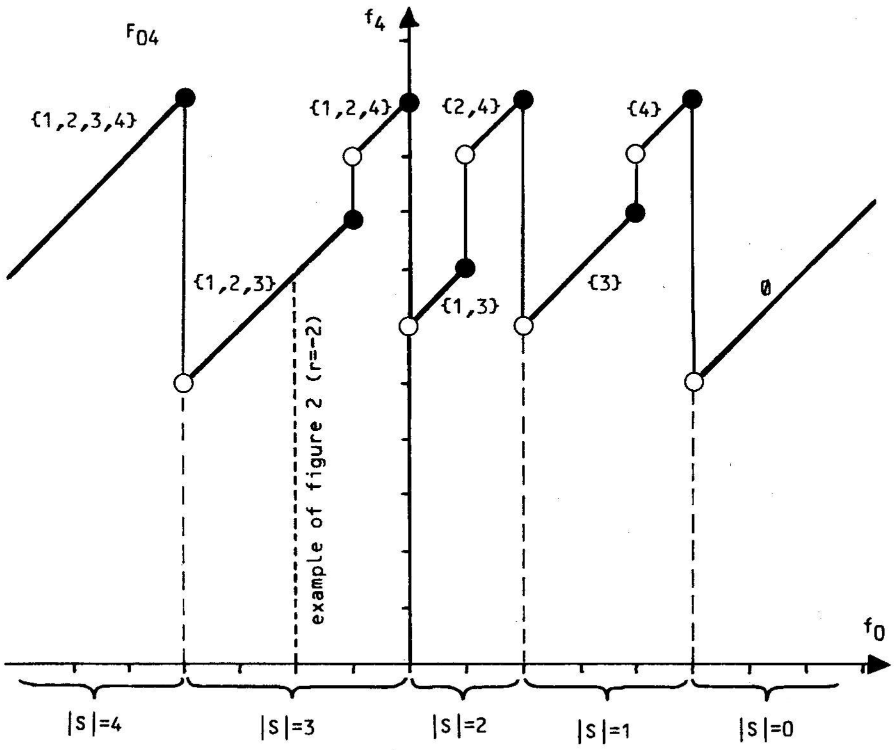


FIGURE 3 (continued)
The functions $F_{03}$, and $F_{04}$.

nate values which occur as left limits (i. e. at the right endpoints of the linear pieces), and at most m different ordinate values which occur as right limits.

(5) The function consists of at most m+1 strictly increasing portions, and the cardinality of the solution set S, restricted to $\{i+1,i+2,\ldots,j\}$, is constant on each portion (cf. figure 3d).

(6) Each of these portions consists of at most m different linear pieces.

PROOF: Properties (1) and (2) are immediate. Now we show (5):
It is clear that the cardinality of $S\cap\{i+1,i+2,\ldots,j\}$ decreases monotonically from m to 0 as we move $f_i$ from left to right. Thus the domain of $F_{ij}$ can be decomposed into m+1 intervals where the cardinality is constant. Let's look at such an interval, where the cardinality is t. Since $F_{ij}(f_i)$ is equal to $f_i$ plus the shortest possible total processing time of a schedule for a subset of the jobs $\{i+1,i+2,\ldots,j\}$ with cardinality t starting at the release date $f_i$, and since any feasible schedule with release date $f_i$ is also feasible for all release dates $f_i'<f_i$, it follows that $F_{ij}$ is strictly increasing in the interval.

Properties (3) and (4) will be shown by induction on m: We conclude from $F_{i,i+m-1}$ to $F_{i,i+m}$. For m=1 the assertion is obvious (cf. figure 3a).
For m>1, $F_{i,i+m}$ is obtained from $F_{i,i+m-1}$ by applying the function $F_{i+m-1,i+m}$, i. e. by adding $d_{i+m}$ to all function values $\leq s_{i+m}$ and leaving the rest as it is (cf. figure 3b, c, and d). New breakpoints can only appear where the function $F_{i,i+m-1}$ crosses the horizontal line at $s_{i+m}$ continuously. Since $F_{i,i+m-1}$ has at most m monotonically increasing portions, this can happen at most m times, and at most m new breakpoints can appear. This implies (3).
The ordinate values at the new breakpoints are $s_{i+m}$ as right limits and $s_{i+m}+p_{i+m}$ as left limits. The ordinate values of $F_{i,i+m}$ at the old breakpoints can be obtained from the ordinate values of $F_{i,i+m-1}$ by adding $p_{i+m}$ to those which are $\leq s_{i+m}$. Thus the number of different ordinate values at the old breakpoints does not increase, and there is one (possibly) new ordinate value as left limit and one as right limit.
Property (6) follows from (5) and (4). ■

Remark 1: This theorem requires that the values $s_i$ and $p_i$ have been preprocessed, otherwise the argument leading to (5) would not be valid. If $s_i$ and $p_i$ can have arbitrary values, the function $F_{ij}$ defined by part B of the algorithm can have an exponential number of pieces.

Remark 2: From figure 3 it can be seen that some elements are dropped from the solution set S and exchanged against others as r moves from right to left, and thus S does not monotonically grow. This is the false assumption on which the algorithm of Dekel and Sahni [1983a] was based.

## 3. DESCRIPTION OF THE ALGORITHM

The algorithm is based on the binary tree method of Dekel and Sahni [1983a]. The idea is to compute $F_{0n}$ by decomposing it into two "equal-sized" parts $F_{0j}$ and $F_{jn}$, which can be computed in parallel and then computing the composition of these functions. $F_{0j}$ and $F_{jn}$ are again computed by parallel decomposition and so on recursively. As a conceptual aid, we draw a complete binary tree with n leaves (cf. figure 4). The leaves are associated with the functions $F_{i-1,i}$, i=1,2,...,n, in that order, and the common predecessor of the nodes which are associated with $F_{ij}$ and $F_{jk}$, respectively, is associated with $F_{ik}$. The root is associated with $F_{0n}$.

After preprocessing the data, we proceed from the leaves to the root, composing the functions of the sons to the functions of the father as we go. Then we evaluate the function $F_{0n}$ at the point r and proceed from the root to the

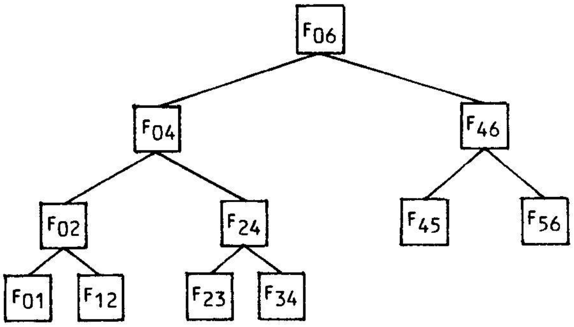


FIGURE 4
The complete binary tree for n=6 jobs.

leaves, computing the values $f_i$, $i=0,\dots,n$. This is done as follows:

> Assume that we are in the node associated with $F_{ik}$, and we have already computed $f_i$, and this node has two sons which are associated with $F_{ij}$ and $F_{jk}$. Then we can compute $f_j:=F_{ij}(f_i)$. Now we know $f_i$ and $f_j$ and we can go down one level in the tree.

Finally, for each i, we know $f_{i-1}$ and we can determine whether job i occurs in the optimal solution. As a last step we have to collect this information and compute the start times for all jobs that are scheduled.

## 4. THE PARALLEL IMPLEMENTATION

We shall repeatedly need the following basic result of Dekel and Sahni [1983a], which is the standard example for the application of the binary tree method.

LEMMA: Given a sequence $(a_1,a_2,\dots,a_m)$ of m numbers, the partial sums sequence $(S_1,\dots,S_m)$ with $S_i=a_1+a_2+\dots+a_i$ can be computed in $O(\log m)$ time with m processors.

We represent each function $F_{ij}$ by an array of at most $(j-i+1)(j-i)/2+1$ triples $(l,u,c)$ sorted by the $l$ component. A triple $(l,u,c)$ has the following meaning:
"If $\mathbf{l}<f_i\leqq\mathbf{u}$, then $F_{ij}(f_i)=f_i+\mathbf{c}$."
($l$ may be $-\infty$, and u may be $\infty$.)

### 4.1. Overview of the parallel algorithm

The preprocessing step consists of sorting the jobs by due-dates and computing the $s_i$ values. Sorting can for example be accomplished by the simple enumeration sorting scheme of Muller and Preparata [1975] with $n^2$ processors in $O(\log n)$ time. The computation of one $s_i$ value can be done with n processors in $O(\log n)$ time by computing partial sums and one parallel subtraction and minimum-finding. Thus the computation of all $s_i$ values takes $O(\log n)$ time on $n^2$ processors.

The composition of two functions $F_{ij}$ and $F_{jk}$ in proceeding from the leaves to the root is the hardest step and is discussed later.

The storage needed for storing the function $F_{ij}$ is $O((j-i)^2)$, and thus a total storage of $O(n^2)$ is enough to store all functions.

In each node of the tree, the determination of $f_j=F_{ij}(f_i)$ can be done in constant time with $(j-i)^2$ processors given the function $F_{ij}$ as a list of sorted triples. Thus the total processor requirement while going from the root to the leaves is at most $n^2$ at each level of tree, and the total time is $O(\log n)$.

Computing the start times for each job can again be accomplished by computing partial sums with n processors in $O(\log n)$ time.

Thus the algorithm, exclusive of the upward pass thru the tree, can be implemented with $n^2$ processors in $O(\log n)$ steps.

## 4.2. Implementation of the composition of two functions

First we give a very simple implementation of the composition of two functions requiring $n^4$ processors. Let $(P_1,P_2,...)$ and $(Q_1,Q_2,...)$ be the lists associated with the functions $F_{ij}$ and $F_{jk}$. Then we provide a processor (a,b) for every pair of triples $P_a=(l_1,u_1,c_1)$ and $Q_b=(l_2,u_2,c_2)$ and let this processor perform the following action.

```
    **if** $l_1+c_1<u_2$ **and** $l_2<u_1+c_1$ **then**
        (The image interval of $F_{ij}$, $(l_1+c_1,u_1+c_1]$, and the domain interval of
        $F_{jk}$, $(l_2,u_2]$, intersect.)
        set up a new triple (l,u,c) for the composed function $F_{ik}$, with:
            $l := \max\{l_1,l_2-c_1\}$;
            $u := \min\{u_1,u_2-c_1\}$;
            $c := c_1+c_2$.
        (We say that the triples $(l_1,u_1,c_1)$ and $(l_2,u_2,c_2)$ have **generated** the
        triple (l,u,c).)
```

Then we have to collect the new triples into one sorted array. We do this by setting $q_{ab}=1$ if a new triple has been set up in processor (a,b) and $q_{ab}=0$ otherwise, and computing the partial sums sequence of the sequence $(q_{11},q_{12},...,q_{21},q_{22},...)$. The (a,b) entry in this sequence is the position where the triple generated in processor (a,b) has to go in the sorted array of triples for the composed function. Because of theorem 2, property (3), at most $(k-i+1)^2/2$ new triples have been generated.
In total, one composition of functions takes $O(\log n)$ time.

Now we show how the processor requirement to compose two functions $F_{ij}$ and $F_{jk}$ can be reduced to $n^2$. The idea is to allocate processors only to those pairs (a,b) of intervals which will generate an interval of $F_{ik}$. We proceed as follows:

1. In step two, we want to allocate one processor to each triple of $F_{ik}$ that will be generated. For each triple (l,u,c) of $F_{ij}$ we find the indices of the intervals of $F_{jk}$ in which l+c and u+c lie by looking them up in the sorted sequence of triples of $F_{jk}$. One plus the difference between these indices is the number of intervals of $F_{jk}$ which the interval $(l+c,u+c]$ intersects, i. e. the number of triples of $F_{ik}$ which are generated with the triple (l,u,c). This is the number of processors that will be allocated to the triple (l,u,c).
This step takes $O(\log n)$ time and $(j-i)^2$ processors.

2. Then we calculate the partial sums sequence of the numbers computed in step one and use this sequence to allocate one processor to each pair of triples of $F_{ij}$ and $F_{jk}$ which will generate a triple of $F_{ik}$.
The computation of partial sums takes $(j-i)^2/2$ processors and $O(\log(j-i))$ time. Because of theorem 2, property (3), at most $(k-i+1)^2/2$ processors are needed.

3. As described above for the simple algorithm, we generate the triples (l,u,c) of $F_{ik}$. This takes constant time.

Thus we have shown that the composition of two functions $F_{ij}$ and $F_{jk}$ can be computed in $O(\log n)$ time with approximately $(k-i)^2$ processors. This means that, as we go up the tree from the leaves to the root, no more than $n^2$ processors are needed at each level of the tree, and the total time from the leaves to the root is $O(\log^2 n)$.

Finally we remark that the same algorithm complexity can be achieved without introducing read conflicts. This involves setting up binary trees through which each data item that is needed in several processors is broadcast. To allow this, the data structure for representing the function $F_{ij}$ must be extended. Since the resulting algorithm is complicated and technical, it is not included in the paper.

## 5. CONCLUSION

We have shown that the scheduling problem of maximizing the number of jobs scheduled in time if n jobs with different processing times and due-dates are given can be solved on $n^2$ processors in $O(\log^2 n)$ time and $O(n^2)$ space. In view of the sequential complexity of $O(n \log n)$ steps this result, which nevertheless requires a thorough analysis of the problem, can only be taken as a first step towards a better or optimal solution. Perhaps the appropriate approach to such problems is not the parallelization of good sequential algorithms, especially in this case where the sequential algorithm looks almost hopelessly sequential at first glance, but searching from scratch for completely different solutions which are more appropriate for a parallel implementation.

## REFERENCES

Dekel, E., and S. Sahni [1981]
Parallel scheduling algorithms.
In: Proc. 1981 Int. Conf. Parallel Processing (M. T. Liu, J. Rothstein, eds.), IEEE Computer Society Press, pp. 350-351.

Dekel, E., and S. Sahni [1983a]
Binary trees and parallel scheduling algorithms.
IEEE Transactions on Computers **C-32**, 307-315.

Dekel, E., and S. Sahni [1983b]
Parallel scheduling algorithms.
Operations Research **31**, 24-49.

Dekel, E., and S. Sahni [1984]
A parallel matching algorithm for convex bipartite graphs and applications to scheduling.
J. Parallel Distr. Computing **1**, 185-205.

Moore, J. M. [1968]
An n job, one machine sequencing algorithm for minimizing the number of late jobs.
Management Science **15**, 102-109.

Muller, D. E., and F. P. Preparata [1975]
Bounds to complexities of networks for sorting and for switching.
J. Assoc. Comput. Mach. **22**, 195-201.