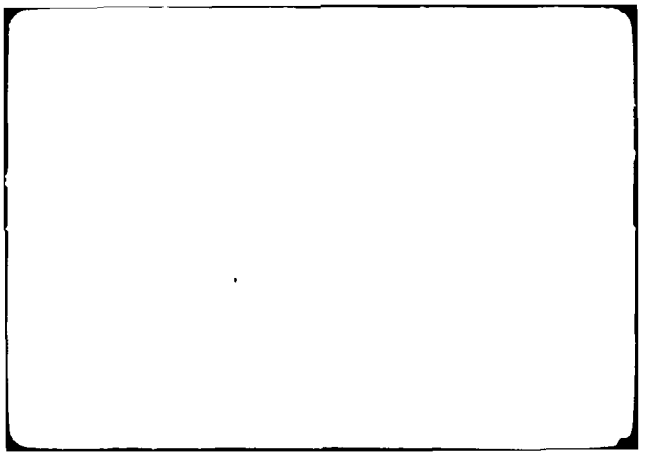
AD-A103 796 MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE F/G 12/2
PARALLEL SCHEDULING ALGORITHMS.(U)
MAR 81 E DEKEL, S SAHNI N00014-80-C-0650
UNCLASSIFIED TR-81-1 NL

END
DATE FILMED
10-81
DTIC



Computer Science Department
136 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

(9) Technical rpt.

(14) TR-81-1

(15) N00014-80-C-0647

(6)

Parallel Scheduling Algorithms,

(10) by
Eliezer Dekel and Sartaj Sahni

Technical Report 81-1

(11) March 1981

(12) 59

Cover design courtesy of Ruth and Jay Leavitt

412936

Parallel Scheduling Algorithms*

Eliezer Dekel and Sartaj Sahni
University of Minnesota

Abstract:

We obtain fast parallel algorithms for several scheduling problems. Some of the problems considered are: scheduling to minimize the number of tardy jobs; job sequencing with deadlines; scheduling to minimize earliness and tardiness penalties; channel assignment; and minimizing the mean finish time. The shared memory model of parallel computers is used.

Keywords and Phrases: parallel algorithm, complexity, shared memory model, mean finish time, earliness, tardiness, deadline.

* This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

1. Introduction

With the continuing dramatic decline in the cost of hardware, it is becoming feasible to economically build computers with thousands of processors. In fact, Batcher [3] 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 techniques and algorithms readily adapt to the other models (eg: multiple instruction stream multiple data stream (MIMD) and data flow models). SIMD computers have the following characteristics:

(1) They consist of p processing elements (PEs). The PEs are indexed $0,1, \ldots, p-1$ 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 PEs 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 PEs that are to perform an instruction. Only the enabled PEs will perform the instruction. The remaining PEs will be idle. All enabled PEs execute the same instruction. The set of enabled PEs can change from instruction to instruction.

While several SIMD models have been proposed, in this paper we shall deal explicitly with only the shared memory model (SMM). In this model, there is a large common memory that is shared by all the PEs. It is assumed that any PE can access any word of this common memory in O(1) time. When two or more PEs access the same word simultaneously, we shall say that a conflict has occured. If all the PEs (at least two) that simultaneously access the same word wish to write in it, it is called a write conflict. If all wish to read, then it is a read conflict. Write conflicts may be permitted so long as all the PEs wish to write the same piece of information. As far as our discussion here is concerned, no read or write conflicts are allowed. A

description of some of the other SIMD models can be found in [5].

Most algorithmic studies of parallel computation have been based on the SMM (1,2,4,7,10,11,19,21,22). Parallel matrix and graph algorithms for the SMM have been developed by Agerwala and Lint [1], Arjomandi [2], Csanky [4], Eckstein [7], Hirschberg, Chandra, and Sarwate [10], and Savage [22]. Hirschberg [11], Muller and Preparata [19], and Preparata [21] have considered the sorting problem for SMMs. The results of Muller and Preparata [19] and Preparata [21] will be made use of in this paper. In these two papers, it is shown that n numbers can be sorted in $O(\log n)$ time if n^2 PEs are available, and in $O(\log^2 n)$ time when n PEs are available.

Dekel and Sahni [6] develop a design technique for parallel algorithms that is based on binary computation trees. This design technique is illustrated using several examples from scheduling theory. Some of the scheduling problems considered by them are:

P1: Schedule many machines to minimize maximum lateness when all jobs have a processing time $p_i=1$.
P2: Schedule one machine to minimize maximum lateness. Preemptions are permitted.
P3: Schedule one machine to minimize the number of tardy jobs.
P4: The job sequencing with deadlines problem.

The complexity of their parallel algorithms for all the above problems is $O(\log^2 n)$.

When measuring the effectiveness of a parallel algorithm, one needs to consider both its complexity as well as its cost in terms of the number of PEs used. The effectiveness of processor utilization (EPU) is defined with respect to a parallel algorithm and the fastest known sequential (i.e., single processor) algorithm for the same problem. Let P be a problem and A a parallel algorithm for P. We define:

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

The algorithm of [6] for problem P1 above uses n/2 PEs and has a complexity of $O(\log^2 n)$. The fastest sequential algorithm known for this problem is due to Horn [4] and runs in $O(n \log n)$ time. So, the EPU of the parallel algorithm of [6] for P1 is $O(n\log n/(n\log^2 n)) = O(1/\log n)$.

The best EPU one can hope for is O(1). Few parallel algorithms achieve this EPU. Dekel and Sahni [6] present some algorithms that do. One that we shall need here is for the partial sums problem. We are given n numbers $a_1, a_2, \ldots, a_n$ and are required to compute $A_j = \bigoplus_{i=1}^{j} a_i$, $1 \le j \le n$, where $\oplus$ is any asscoiative operator (eg. max, min, +, *). Their algorithm run in O(logn) time and uses n/logn PEs.

In this paper, we consider several scheduling problems. Fast parallel algorithms are obtained for each. In each case, the complexity analysis is carried out on the assumption that as many PEs as needed are available. This is in conformance with the assumption made in almost all the research work done on parallel computing. This assumption is of course unrealistic. A parallel algorithm will eventually be run on a machine with a finite number (say k) of PEs. It should be easy to see that all our algorithms are easily adapted to the case of k PEs. If our algorithm has complexity O(g(n)) using f(n) PEs, then with k PEs, $k < f(n)$, its complexity is O(g(n)f(n)/k). We shall continue with tradition, and explicitly analyse our algorithms only for the case when as many PEs as needed are available.

In Sections 2 and 3, we consider two relatively simple examples. The first of these is to minimize the finish time when m identical machines are available. The second example is to minimize the mean finish time when m uniform machines are available. In Sections 4, 5, 6, and 7, we respectively, consider the following problems:

(i) minimize the number of tardy jobs when $p_i = 1$ $1 \le i \le n$ and 1 machine is available.
(ii) job sequencing with deadlines.
(iii) schedule one machine to minimize the maximum earliness and tardiness penalties.

and

(iv) channel assignment.

2. Minimum Finish Time

When preemptions are permitted, a minimum finish time schedule for m machines is efficiently obtained using Mc Naughton's rule [17]. Let $p_1, p_2, \ldots, p_n$ be the processing times of the n jobs. The finish time, f, of an optimal preemptive schedule is given by:

$$f = \max\{ \max_{1 \le i \le n} \{p_i\}, \frac{1}{m} \sum_{i=1}^{n} p_i \}$$

Using f, the optimal schedule may be constructed in O(n) time [17]. Job 1 is scheduled on machine 1 from 0 to p_1 and job 2 from p_1 to $\min\{p_1+p_2, f\}$. If $p_1 + p_2 > f$, then

the remainder of job 2 is done on machine 2 starting at time ∅. If $p_1 + p_2 \leq f$, then job 3 is scheduled on machine 1 from p_1+p_2 to $\min\{p_1 + p_2 + p_3, f\}$; etc.

Using the parallel algorithms of [6], $\max\{p_i\}$ and $\sum_{i=1}^{n} p_i$ may be computed in O(logn) time with n/logn PEs. To obtain the actual schedule, we also need $A_i = \sum_{j=1}^{i} p_j$, $1 \leq i \leq n$. As mentioned in Section 1, all the A_is can be computed in O(logn) time using n/logn PEs ([6]). Let $A_\emptyset = \emptyset$. Each job i can now determine its own processing assignment by using the following rule:

```
x ← ⌈A_{i-1}/f⌉ * f - A_{i-1}
case
   :x=∅ : schedule job i on machine ⌈A_i/f⌉ from ∅ to p_i
   :x>p_i: schedule job i on machine ⌈A_i/f⌉ from
           f-x to f-x+p_i
   :else: schedule job i on machine ⌈A_i/f⌉ from ∅ to
          p_i-x
end case
```

One may verify that x gives the amount of processing time left on the machine $\lceil A_{i-1}/f \rceil$ after job i-1 is finished on that machine.

Example 2.1: Suppose we have 14 jobs with processing times as given in Figure 2.1. Let m=5. f=max{7, 50/5}=10. Figure 2.1 gives the A_i and x values for each job. The schedule obtained is given in Figure 2.2.□

Job	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P_i	5	3	1	2	7	4	1	5	2	4	6	1	6	3
A_i	5	8	9	11	18	22	23	28	30	34	40	41	47	50
x	0	5	2	1	9	2	8	7	2	0	6	0	9	3

Figure 2.1

If we have n PEs, all the machine assignments can be computed in O(1) time. However, using only n/logn PEs, these assignments may be obtained in O(logn) time (i.e., each PE computes at most $\lceil \log n \rceil$ assignments). So, the overall scheduling algorithm has a complexity of O(logn) and uses n/logn PEs. So, its EPU is O(n/(logn*n/logn))=O(1).

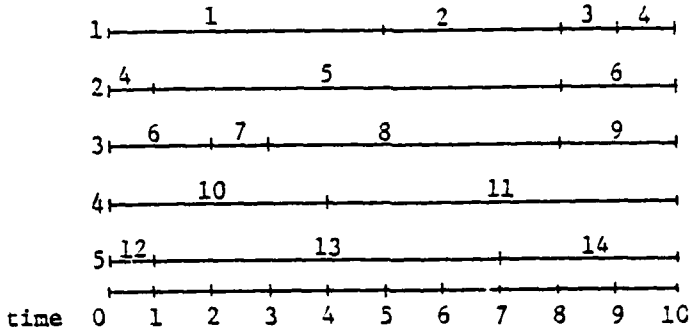


Figure 2.2

3. Minimum Mean Finish Time

A non-preemptive schedule that minimizes the mean finish time of n jobs on m identical machines is obtained by using the LPT rule. By simply using a parallel sorting algorithm, this schedule may be obtained in $O(\log n)$ time with n^2 PEs or in $O(\log^2 n)$ time with n PEs.

Let us consider the case of m uniform parallel machines. Associated with machine i is a speed s_i. It takes machine i, p_i/s_i time units to complete the processing of job i. Horowitz and Sahni [13] present an $O(n \log mn)$ algorithm that constructs a minimum mean finish time schedule for this case. Their algorithm is reproduced in Figure 3.1. This algorithm assumes that the speeds and processing times have been normalized and sorted such that $s_1=1 \leq s_2 \leq \cdots \leq s_m$ and $p_1 \leq p_2 \leq \cdots \leq p_n$.

By examining this algorithm, we see that another way to obtain an optimal schedule is to sort the mn numbers i/s_j, $1 \leq i \leq n$, $1 \leq j \leq m$ into nondecreasing order. Let the resulting sequence be $a_1, a_2, a_3, \ldots, a_{mn}$. If a_i corresponds to q/s_j, then job $n+1-i$ is scheduled on machine j and there are $q-1$ jobs following it on that machine.

This information may be obtained in $O(\log^2 mn)$ time using a parallel sort and mn PEs or in $O(\log mn)$ time with $m^2 n^2$ PEs. If we use the former sort algorithm, the EPU of our parallel algorithm is $O(n \log mn/(mn * \log^2 mn)) = O(1/(m \log mn))$. If the latter sort algorithm is used, the EPU of our scheduling algorithm becomes $O(n \log mn/(m^2 n^2 \log mn)) = O(1/(m^2 n))$. The actual start and finish times for each job can be obtained by later using the partial sums algorithm of [6].

Algorithm MFT

Input: m processors with speeds $1, s_2, \ldots, s_m$, $1 \le s_2 \le \ldots \le s_m$; n jobs initially sorted so that $p_1 \le p_2 \le \ldots \le p_n$ where the times p_i are for processor 1.

Output: Sets R_i $1 \le i \le m$. The jobs in R_i are to be run on processor i in increasing order of their execution times.

for $j \leftarrow 1$ to $m - 1$ do
$R_j \leftarrow \phi$; $i_j \leftarrow 1/s_j$
end for
$R_m \leftarrow \{n\}$; $i_m \leftarrow 2/s_m$
//Note that the above assigns the job with the largest processing time to the fastest processor, m.//
for $k \leftarrow n - 1$ to 1 do
Let u be the largest index such that $i_u = \min_{1 \le j \le m} \{i_j\}$
$i_u \leftarrow i_u + 1/s_u$
end for

end MFT

Figure 3.1

4. Number of Tardy Jobs

Let $J = \{(p_i, d_i) \mid 1 \le i \le n\}$ define a set of n jobs p_i is the processing time of job i and d_i is its due time. Let S be any one machine schedule for J. Job i is *tardy* in the schedule S iff it completes after its due time d_i.

Hodgson and Moore [18] have developed an $O(n \log n)$ sequential algorithm that obtains a schedule that minimizes the number of tardy jobs. Dekel and Sahni [6] present an $O(\log^2 n)$ parallel algorithm to obtain a schedule with the fewest number of tardy jobs. This algorithm uses $O(n)$ PEs and has an EPU of $O(1/\log n)$.

In this section, we shall develop a parallel algorithm for the case when $p_i = 1$, $1 \le i \le n$. This algorithm will have a complexity $O(\log n)$. It will require $O(n^2)$ PEs and thus have an EPU that is $O(1/n)$. While the algorithm of this section has an EPU that is inferior to that of [6], it is faster by a $\log n$ factor. It is interesting to note that the simplification $p_i = 1$, $1 \le i \le n$ does not lead to a corresponding speed up for the sequential case.

The problem of finding a schedule that minimizes the number of tardy jobs is equivalent to that of selecting a maximum cardinality subset U of J such that every job in U can be completed by its due time. Jobs not in U can be scheduled after those in U and will be tardy. A set of jobs U such that every job in U can be scheduled to complete by its due time is called a *feasible set*. It is well known that a set of jobs U is feasible iff scheduling jobs in U in

nondecreasing order of due times results in no tardy jobs (see [14] for eg.).

When $p_i=1$, $1\leq i\leq n$, a maximum cardinality feasible set U can be obtained by considering the jobs in nondecreasing order of due times. The job j currently being considered can be added to U iff $|U|<d_j$. Procedure FEAS(J,b) is a slight generalization. It finds a maximum subset of J that can be scheduled in the interval [0,b]. DONE(i) is set to -1 if job i is not selected and is set to a number greater than 0 otherwise. If DONE(i) > 0, then job i is to be scheduled from DONE(i) - 1 to DONE(i). The procedure itself returns a value that equals the number of jobs selected. The correctness of FEAS is easily established using an exchange argument. Its complexity is O(nlogn) as it takes this much time to order the jobs by due time.

```
line Procedure FEAS(J,n,b)
          //select a maximum number of jobs for processing//
          //in [0,b]  n=|J|//
   1    set J; integer n,b; global DONE(1:n)
   2    sort J into nondecreasing order of due times
   3    DONE(1:n) ← -1  //initialize//
   4    j ← 0
   5    for i ← 1 to n do
   6       case
   7       :j>b: return(j) //interval full//
   8       :j<d_i: //select i// j ← j+1; DONE(i) ← j
   9       end case
  10    end for
  11    return(j)
  12 end FEAS
```

Figure 4.1

Let J be a set of n unit processing time jobs. Let D(i), $1\leq i\leq k$ be the distinct due times of the jobs in J. Assume that D(i) < D(i+1), $1\leq i\leq k$. Let n(i) be the number of jobs in J with due time D(i), $1\leq i\leq k$. Clearly, $\sum n(i)=n$. Let D(0)=0 and n(0)=0. Define F(i) to be the value of j when procedure FEAS (Figure 4.1) has just finished considering all jobs in J with due time at most D_i. It is evident that:

$$F(0) = D(0) = 0$$
$$F(i) = \min\{F(i-1)+n(i), D(i), b\},\ 1\leq i\leq k \qquad (4.1)$$

Expanding the recurrence (4.1), we obtain:

$$F(1) = \min\{D(0)+N(1),\ D(1),\ b\}$$
$$F(2) = \min\{F(1)+n(2),\ D(2),\ b\}$$
$$= \min\{D(0)+n(1)+n(2), D(1)+n(2), b+n(2), D(2), b\}$$

$$= \min\{D(0)+n(1)+n(2), D(1)+n(2), D(2), b\}$$

$$F(3) = \min\{D(0)+n(1)+n(2)+n(3), D(1)+n(2)+n(3), D(2)+n(3), D(3), b\}$$

. . .

And, in general

$$(4.2)\quad F(m) = \min\{\min_{1\le i\le m}\{D(i)+\sum_{q=i+1}^{m} n(q)\}, b\} \quad 0\le m\le k$$

The maximum number of jobs in J that can be scheduled in $[0,b]$, $b>0$, so that none is tardy is $F(k)$. $F(k)$ may be efficiently computed, in parallel as follows. Let the due times of the n jobs in J be $d(1), d(2), \ldots, d(n)$. Let $d(0)=0$. We may assume that $d(i)>0$, $1\le i\le n$. The computation steps are:

Step 1: sort $d(1:n)$ into nondecreasing order.

Step 2: determine the points $r(0), \ldots, r(k-1)$ in $d(0:n)$ where the due times change I.e. $r(i) < r(i+1), 1\le i<k$ and $d(r(i)) \ne d(r(i)+1)$. Let $r(k)=n$. Clearly, $r(0)=0$, and $n(i)=r(i)-r(i-1)$ and $D(i) = d(r(i))$, $1\le i\le k$; $D(0)=a$.

Step 3: since $D(i) + \sum_{q=i+1}^{k} n(q) = D(i)+n-r(i)$, we compute $F(k)=\min\{n+ \min_{0\le i\le k}\{D(i)-r(i)\}, b\}$ []

Example 4.1 Figure 4.2(a) gives the due times of a set J of 15 jobs. In Figure 4.2(b), the jobs have been ordered by due times. The points at which the due times change are shown by heavy lines. We see that k=6; $r(0:6) = (0, 3, 7, 8, 9, 13, 15)$ and $D(0:6) = (0, 2, 3, 5, 8, 9, 11)$. So, $n+ \min_{0<i\le k}\{D(i)-r(i)\} = 15 + \min\{0, -1, -4, -3, -1, -4, -4\} = 15-4=11$. If $b \ge 11$, then the maximum number of nontardy jobs is 11. []

With n^2 PEs, step 1 can be carried out in $O(\log n)$ time. (see [19] and [21]). Using n-1 PEs, the boundary points can be found in $O(1)$ time. PE(i) simply checks to see if $d(i)<d(i+1)$, $1\le i\le n-1$. If so, then i is a boundary point. 0 and n are also boundary points. The boundary points have now to be moved into memory positions $r(0), r(1), \ldots, r(k)$. This can be done in $O(\log n)$ time using n PEs and the data concentration algorithm of [20]. Another data concentration step moves $d(r(0)), d(r(1)), \ldots, d(r(k))$ into $D(0), D(1), \ldots, D(k)$. Using k+1 PEs, $D(i)-r(i)$, $0\le i\le k$ can be computed in $O(1)$ time. $\min\{D(i)-r(i)\}$ can be obtained in $O(\log k)$ time using the binary tree computation model of [6] (Figure 4.3 shows this for our example.) As explained in [6], only $O(k/\log k)$ PEs are needed for this; but using $k/2$ PEs is

job	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d_i	3	2	5	2	9	3	11	9	11	3	8	9	3	9	2

(a) input set of jobs.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
job	0	2	4	15	1	6	10	13	3	11	5	8	12	19	7	9
d	0	2	2	2	3	3	3	3	5	8	9	9	9	9	11	11

(b) jobs sorted in nondecreasing order of due time

Figure 4.2

faster). $F(k)$ can now be computed using an additional $O(1)$ time. The overall complexity is therefore $O(\log n)$ and n^2 PEs are used. The EPU of the above algorithm is $O((n\log n/(\log n * n^2)) = O(1/n)$.

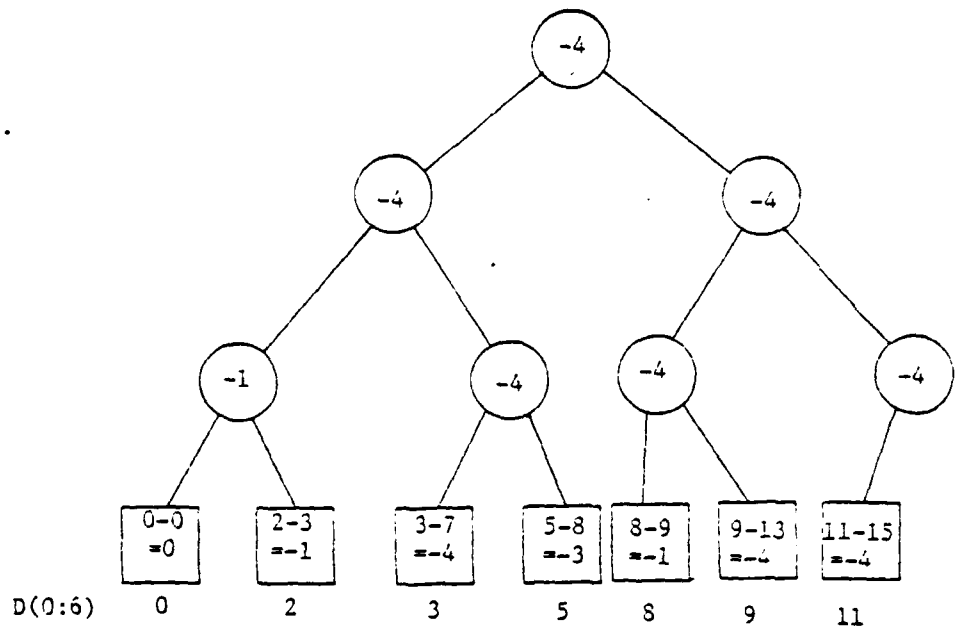


Figure 4.3

We have seen how to determine the maximum number of nontardy jobs. In some applications (see the next section),

this is adequate. To obtain the actual schedule, we may proceed as follows. First, modify procedure FEAS by adding the line:

8.1 :else: DONE(i) ← j

and by deleting line 7.

It is easy to see that job i is completed at time DONE(i) iff DONE(i)≤b and DONE(i-1) ≠ DONE(i), 1≤i≤n. For the modified algorithm, we see that:

$$\begin{aligned} &DONE(0) = 0 \\ (4.3)\quad &DONE(i) = \min\{DONE(i-1)+1, d_i\},\ 1 \le i \le n \end{aligned}$$

Solving (4.3), we obtain:

$$(4.4)\quad DONE(i) = \min_{0 \le j \le i} \{d_j + i - j\},\ 1 \le i \le n$$

DONE(i), 1≤i≤n may be computed in O(logn) time using n^2 PEs (though $n^2/\log n$ are sufficient) and the binary computation tree model (see [6] and Figure 4.3). Since the initial sort takes O(logn) time and requires n^2 PES, the overall time complexity is O(logn) and the EPU is O(1/n). From DONE(i), the schedule is easily obtained.

Example 4.2: For the sorted data of Example 4.1, we obtain DONE(0:15)=(0, 1, 2, 2, 3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11). So the set of non tardy jobs in [0,b], b≥11 is {2, 4, 1, 3, 11, 5, 8, 12, 14, 7, 9}. By concentrating these to the left, we obtain the permutation (2, 4, 1, 3, 11, 5, 8, 12, 14, 7, 9, 15, 6, 10, 13) which represents an optimal schedule. []

5. Job Sequencing With Deadlines

In this problem, we are given a set J of n jobs. Associated with job i is a profit z_i and a due time d_i, 1≤i≤n. Every job has a processing requirement of one unit. If job i is completed by time d_i, then a profit z_i, z_i>0 is made. If job i is not completed by the time d_i, then nothing is earned. We wish to select a feasible subset of J that yields maximum return (recall that R is a feasible subset iff all jobs in R can be scheduled to complete on time).

One way to find a feasible subset R of J that gives maximum return is:

```
Step 1: sort J into nonincreasing order of z_i
Step 2: R ← {1}
        for i ← 2 to n do
          if R U{i} is feasible then R ← RU{i}
        end for
```

Figure 5.1

A correctness proof of the above procedure may be found in [14]. It is also possible to implement the above scheme by a sequential algorithm of complexity O(nlogn). For the parallel version, we reduce the job sequencing with deadlines problem into 2n independent feasibility problems. First, we note that if R1 and R2 are feasible subsets of J and if R1 is one with maximum return, then $|R2| \leq |R1|$.

Theorem 5.1: Let A be a feasible subset of J that yields maximum return. Let B be any feasible subset of J. $|B| \leq |A|$.

Proof: Since A and B are feasible subsets, they can respectively be scheduled in $[\emptyset, |A|]$ and $[\emptyset, |B|]$ in such a manner that no job is tardy. Consider such a scheduling SA of A and SB of B. Consider a job i that is in both A and B. If i is scheduled earlier in SA than in SB, we may change SA by moving i to the slot it is scheduled in B. This would require moving the job (if any) scheduled in this slot in SA to the position previously occupied by i (see Figure 5.2(a)). A similar transformation may be made if i is scheduled later in SA than in SB (see Figure 5.2(b)).

By performing the above transformation on all jobs in A n B, we obtain schedules SA' and SB' that contain no tardy jobs. In addition, jobs in A n B are scheduled in the same slots in SA' and SB'.

If $|B| > |A|$, then there must be job j scheduled in SB' in a slot that is empty in SA'. Also, $j \notin A$. By adding j to A, we clearly obtain a feasible set with return larger than that obtained from A. This contradicts the assumption on A. So, $|B| \leq |A|$. []

From the sequential algorithm for the job sequencing problem and Theorem 5.1, we may derive a parallel algorithm. Let $T1(i) = \{j | z_j > z_i \text{ or } (z_j = z_i \text{ and } j < i)\}$ and $T2(i) = T1(i) \cup \{i\}$. Consider a schedule for T1(i) that has the fewest number of tardy jobs. Let x(i) be the number of non-tardy jobs in this schedule. Let y(i) be the corresponding number for T2(i). From our discussions, it follows that job i will be included in R (Figure 5.1) iff $y(i) > x(i)$. Hence, R may be obtained by computing x(i) and y(i), $1 \leq i \leq n$. x(i) and y(i) may be computed using the parallel algorithm for F(k) described in Section 4. From R, the optimal

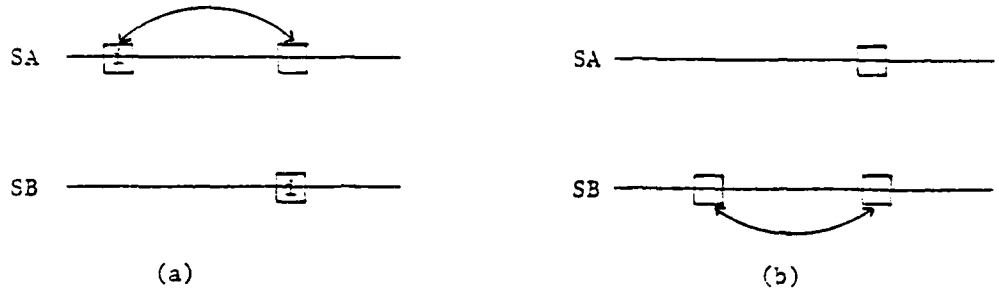


Figure 5.2: Lining up common jobs.

schedule is obtained by scheduling the jobs in R first, in order of due times; and then scheduling the remaining jobs in any order. This construction can be carried out by first concentrating the jobs in R and then sorting them by due times.

Example 5.1: Figure 5.3(a) shows an example job set with 12 jobs. These have been ordered by due times in Figure 5.3(b). Figure 5.4 gives T2(i), $1 \le i \le n$. The number of non-tardy jobs in the optimal schedules for T1(i) and T2(i) is respectively given in (x(i),y(i)). It also tells if job i is to be included in R. R is seen to be {1, 3, 5, 6, 8, 9, 11, 12}. These jobs may be concentrated to one end to obtain Figure 5.5. This gives the optimal schedule. []

i	1	2	3	4	5	6	7	8	9	10	11	12
d_i	2	6	6	2	6	2	6	6	7	3	7	8
z_i	85	55	65	80	70	35	60	80	75	60	85	60

(a)

i	1	4	6	10	2	3	5	7	8	9	11	12
d_i	2	2	2	3	6	6	6	6	6	7	7	8
z_i	85	80	85	60	55	65	70	60	80	75	85	60

(b)

Figure 5.3

T2(1)	i	1	
z_1=85	d_i	2	
		include	(Ø,1)
T2(2)	i	1 4 6 1Ø 2 3 5 7 8 9 11 12	
z_2=55	d_i	2 2 2 3 6 6 6 6 6 7 7 8	
		reject	(8,8)
T2(3)	i	1 4 6 3 5 8 9 11	
z_3=65	d_i	2 2 2 6 6 6 7 7	
		include	(5,7)
T2(4)	i	1 4 6 11	
z_4=8Ø	di	2 2 2 7	
		reject	(3,3)
T2(5)	i	1 4 6 5 8 9 11	
z_5=7Ø	d_i	2 2 2 6 6 7 7	
		include	(5,6)
T2(6)	i	1 6	
z_6=85	d_i	2 2	
		include	(1,2)
T2(7)	i	1 4 6 3 5 7 8 9 11	
z_7=6Ø	d_i	2 2 2 6 6 6 6 7 7	
		reject	(7,7)
T2(8)	i	1 4 6 8 11	
z_8=8Ø	d_i	2 2 2 6 7	
		include	(3,4)
T2(9)	i	1 4 6 8 9 11	
z_9=75	d_i	2 2 2 6 7 7	
		include	(4,5)
T2(1Ø)	i	1 4 6 1Ø 3 5 7 8 9 11	
$z_{1Ø}$=6Ø	d_i	2 2 2 3 6 6 6 6 7 7	
		reject	(7,7)
T2(11)	i	1 6 11	
z_{11}=85	d_i	2 2 7	
		include	(2,3)
T2(12)	i	1 4 6 1Ø 3 5 7 8 9 11 12	
z_{12}=6Ø	d_i	2 2 2 3 6 6 6 6 7 7 8	
		include	(7,8)

Figure 5.4

Complexity Analysis

	feasible jobs								late jobs			
i	1	6	3	5	8	9	11	12	4	10	2	7
d_i	2	2	6	6	6	7	7	8	2	3	6	6
w_i	85	85	65	70	80	75	85	60	80	60	55	60

Figure 5.5 The optimal schedule

As far as the complexity is concerned, the initial sort by due times can be done in O(logn) time using n^2 PEs. Next, we need to replicate this sorted data into n copies, one to be used for each T1(i) and T2(i). This replication can be carried out using n^2 PEs and spending O(logn) time (the O(logn) time is needed to avoid read conflicts). Now, the n^2 PEs are divided into n groups of n PEs each. Group i computes T1(i) and then T2(i). T1(i) is obtained by having the jth PE in group i flag job j iff $z_j > z_i$ or ($z_j = z_i$ and $j<i$). Next, the flagged jobs are concentrated in O(logn) time using the n PEs in each group. Note that this concentration preserves the due time ordering. The n PEs in group i next compute $x(i) = F(\kappa_i)$, $1 \le i \le n$. This takes O(logn) time. y(i), $1 \le i \le n$ is computed in a manner similar to that used to obtain x(i).

Having obtained x(i) and y(i), n PEs are used to determine if $y(i) > x(i)$, $1 \le i \le n$. The selected jobs can be concentrated in O(logn) time using these n PEs. The concentration preserves the due time ordering of the selected jobs.

The overall complexity of our parallel algorithm is therefore O(logn). It uses n^2 PEs and has an EPU of O(1/n). This should be contrasted with the algorithm presented by us in [6] for the same problem. That algorithm has a complexity of $O(\log^2 n)$ but uses only O(n) PEs. Thus, its EPU is O(1/logn).

6. Earliness and Tardiness Penalties

Let J be a set of n jobs. Associated with each job is a target start time a_i, a target due time b_i, and a processing time p_i. Any one machine schedule S for J may be denoted by a vector $(s_1, s_2, \ldots, s_n)$ where s_i is the start time of job i. Schedule S is *admissable* iff $s_i \ge s_{i-1} + p_{i-1}$, $2 \le i \le n$. The *computation time* c_i of job i is $s_i + p_i$. The earliness e_i and tardiness t_i of job i are given by:

$$e_i = \max\{0, a_i - s_i\}$$
$$t_i = \max\{0, c_i - b_i\}$$

If job i is early (i.e., $e_i > \emptyset$) then it incurs a penalty $g(e_i)$. If it is tardy (i.e., $t_i > \emptyset$), then it incurs a penalty $h(t_i)$. The objective is to find a schedule S that minimizes the maximum penalty. This problem was first studied by Sydney [23]. He obtained an $O(n^2)$ algorithm for the case when:

(1) $a_i \leq a_j$ implies $b_i \leq b_j$

and

(2) $g(\ )$ and $h(\ )$ are monotone nondecreasing continuous functions such that $g(\emptyset) = h(\emptyset) = \emptyset$.

Our notations and definitions are taken from Sidney's paper. Sidney's $O(n^2)$ algorithm was subsequently improved to O(nlogn) by Lakshminarayan et al. The parallel algorithm we shall develop here is based on the algorithm of Lakasnminarayan et. al [16].

The algorithm of [16] first finds an admissable schedule S using procedure ADMIS (Figure 6.1). This procedure assumes that the jobs are ordered by target start times (i.e., $a_i \leq a_{i+1}$) and within start times by target due times (i.e., $a_i = a_{i+1}$ implies $b_i \leq b_{i+1}$). The maximum lateness, Δ, in S is next computed. If $\Delta = \emptyset$, then S_* is clearly optimal (as $\max\{e_i\} = \max\{t_i\} = \emptyset$). If $\Delta > \emptyset$, then E^* is computed using one of their lemmas. Finally, all the start times in S are decreased by E^*. The new schedule is optimal.

```
Procedure ADMIS (a,p,s,n)
    //jobs are ordered by target start and due times)//
    declare n, a_{1:n}, p_{1:n}, s_{1:n}
    s_1 ← a_1
    for i ← 2 to n do
        s_i ← max{a_i, s_{i-1}+p_{i-1}}
    end for
end ADMIS
```

Figure 6.1

Δ can be computed in O(logn) time using n PEs (see [6]). As described in [16], E^* may be computed in O(1) time using 1 PE. Once E^* has been obtained, n copies of it can be made in O(logn) time using n PEs. Finally, the s_is can be updated in O(1) time using n PEs. Also, the initial ordering of the jobs may be carried out in O(logn) time with n^2 PEs. All that remains, is the computation of the admissable schedule. From Figure 6.1, we see that

$$\begin{aligned} s_1 &= a_1 \\ s_i &= \max\{a_i, s_{i-1} + p_{i-1}\}, \quad 2 \leq i \leq n \end{aligned} \tag{6.1}$$

Expanding the recurrence (6.1), we obtain

$$(6.2)\quad s_i = \max_{1\le j\le i} \{a_j + \sum_{k=j}^{i-1} p_k\}, 1\le i\le n$$

It should be easy to see that using (6.2) and $O(n^3)$ PEs, one can compute all the s_is in O(logn) time. We shall devote the remainder of the section to the development of an O(logn) algorithm that utilizes only n/2 PEs. As we shall see later, $\lceil n/\log n \rceil$ are all that is needed.

For convenience, we shall assume that the jobs are indexed $0,1,\ldots,n-1$ rather than $1,2,\ldots,n$. Before describing the algorithm, we develop some terminology. Let S(0:n-1) be an array. A 2^k-block of S consists of all elements of S whose indices differ only in the 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 iff 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.

Let A(0:n-1) and P(0:n-1) be the target start times and the processing times. Let [i, i+1, i+2,...,r] be the index set for any 2^k-block (a 2^k-block has 2^k indices unless it is the last 2^k-block). With respect to this 2^k-block, we define

$$S(j) = \sum_{q=i}^{j-1} P(q),\ j \text{ is an index in this block}$$

$$(6.3)\quad T(j) = \sum_{q=i}^{r} P(q),\ r \text{ is the highest index in the block}$$

$$Q(j) = \max_{i\le q\le j} \{A(q) + \sum_{t=q}^{j-i} P(t)\},\ j \text{ is a block index}$$

$$U(j) = Q(r) + P(r),\ j \text{ is a block index}$$

For a 2^0-block [i], we have:

$$(6.4)\quad S(i) = 0;\ T(i) = P(i);\ Q(i)=A(i);\ U(i)=A(i)$$

Let X_k = [i,i+1,...,u] and Y = [u+1,...,v] be two sibling 2^k-blocks. Their union Z = [i,i+1,...,v] is a 2^{k+1}-block. Let S, T, Q, and U be the values defined in (6.3) with respect to the 2^k-blocks. Let S', T', Q', and U' be the values defined with respect to the 2^{k+1}-block Z. From (6.3), we see that:

$$(6.5a)\quad S'(j) = \begin{cases} S(j) & \text{if } j \in X \\ S(j)+T(i) & \text{if } j \in Y \end{cases}$$

$$(6.5b)\quad T'(j)=\begin{cases} T(j)+T(u+1) & \text{if } j \leftarrow X \\ T(j)+T(i) & \text{if } j \leftarrow Y \end{cases}$$

$$(6.5c)\quad Q'(j)=\begin{cases} Q(j) & \text{if } j \leftarrow X \\ \max\{Q(j),\ U(i)+S(j)\} & \text{if } j \leftarrow Y \end{cases}$$

$$(6.5d)\quad U'(j)=\ Q'(v)+P(v)$$

One also notes that with respect to the entire $2^{\lceil \log_2 n \rceil}$-block $[0,1,2,3,\ldots,n-1]$,

$$Q(j) = \max_{0 \le q \le j} \{a_q + \sum_{t=q}^{j-1} p(t)\}$$

$$= s_j \text{ of (6.2)}$$

Our strategy is to compute the admissable schedule obtained by procedure ADMIS by using (6.5 a-d). We start with the S, T, Q, and U values for 2^0-blocks as given by (6.4). Next using (6.5 a-d), the S, T, Q and U values for 2^1-blocks are obtained; then for 2^2-blocks, then for 2^3-blocks; etc. Until we have obtained the Q values for the entire $2^{\lceil \log_2 n \rceil}$-block.

Example 6.1: Figure 6.2 gives a set of 10 jobs (indexed 0 through 9). The first row of Figure 6.3 gives the S, T, Q, and U values for the 2^1-blocks; etc. The numbers with arrows give PE assignments. From the bottommost row, we obtain s=(0,3,5,8,12,13,16,20,21,24) as the admissable schedule. []

i	0	1	2	3	4	5	6	7	8	9
P	3	2	2	4	1	3	4	1	3	4
A	0	1	4	8	9	9	15	15	16	17

Figure 6.2 An Example of data set

Let us now proceed to the formal algorithm. In the actual algorithm, processors are assigned to compute the new values of S, T, Q, and U. Assume that the PEs are indexed $0,1,\ldots,\lfloor n/2 \rfloor -1$. With respect to our example of Figure 6.3, when k=0, PE(0) will compute the new values of S(1), T(0), T(1), Q(1), U(0), and U(1); PE(1) will compute S(3), T(2), T(3), Q(3), U(2), and U(3); etc. When k=1, PEs 0 and 1

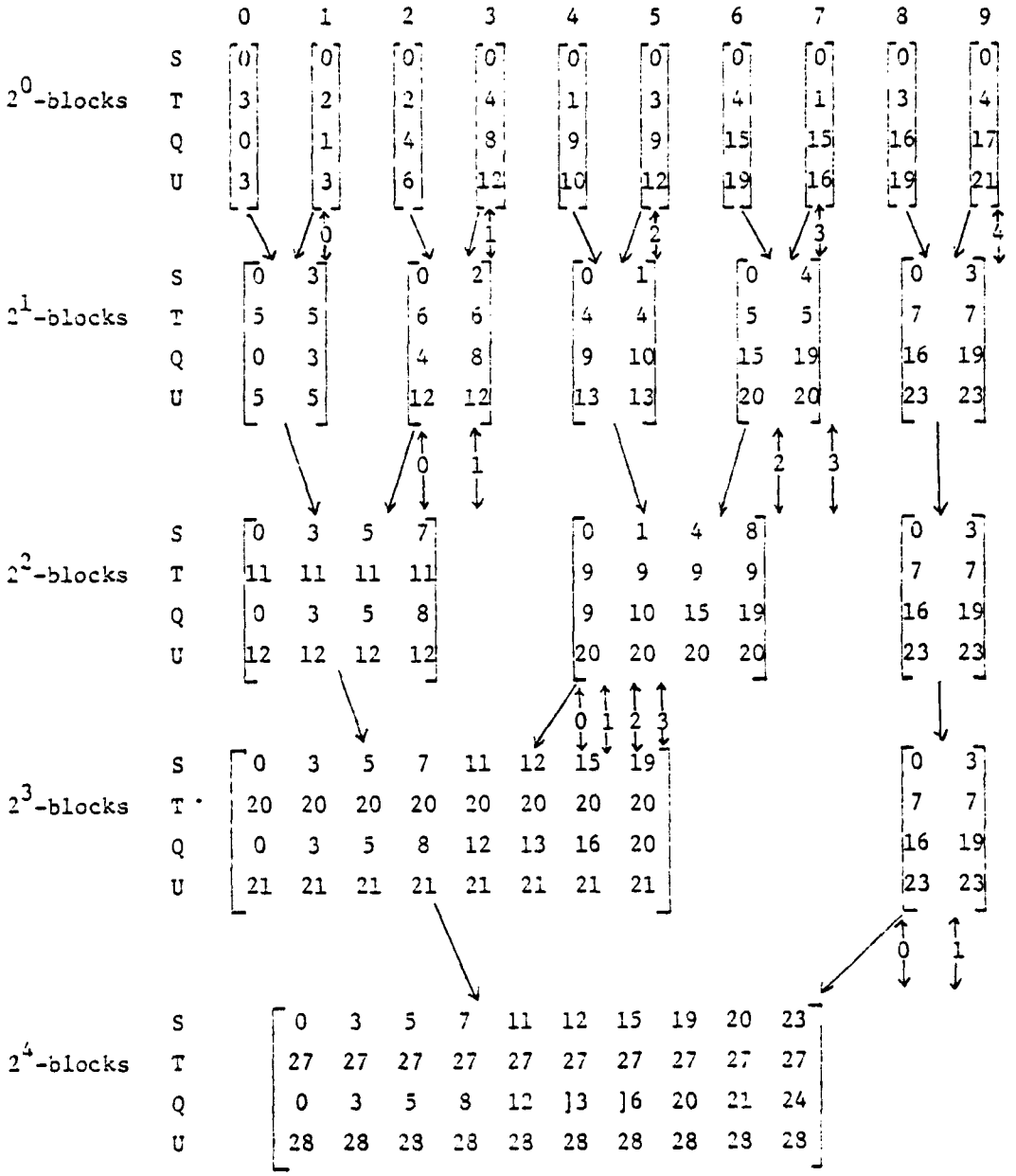


Figure 6.3 Computing the admissable schedule

are both assigned to the new 2^2-block [0,1,2,3], being constructed. PEs 2 and 3 are assigned to the block [4,5,6,7]. PEs 4 and 5 are idle.

Let $\ldots i_3, i_2, i_1, i_0$ be the binary representation of i. The PE assignment rule is obtained by defining the function $f(i,j)=\ldots i_{j+1}, i_j 0\, i_{j-1} \cdots i_0$. When 2^k-blocks are being combined, PE(i) computes $S(f(i,k)+2^k)$, $T(f(i,k))$, $T(f(i,k)+2^k)$, $Q(f(i,k)+2^k)$, $U(f(i,k))$, and $U(f(i,k)+2^k)$ (provided of course that all these indices are less than n). The formal algorithm is given in Figure 6.4. This algorithm mirrors equations (6.5 a-d). Some minor modifications have however been made. Since T( ) is the same for all indices in a 2^k-block, $S(j)+T(i)$ of (6.5a) has been replaced by $S(j)+T(j-2^k)$. Similarly, $T(j)+T(u+1)$ has been replaced by $T(j)+T(j+2^k)$; and $U(i)+S(j-1)$ by $U(j-2^k)+S(j)$. Note that as a result of this change, new T and U values for the right most block may be incorrect (as $j+2^k$ may exceed n-1). This does not affect the outcome of the algorithm as the T and U values of rightmost blocks are never used. One may verify that $\max\{U(j+2^k), U(j)+T(j+2^k)\}=Q'(v)+P(v)$ (Eq. 6.5d). When $k=\lfloor \log n \rfloor -1$, only Q need be computed.

```
procedure PADMIS(A,P,s,n)
   //obtain the admissable schedule(s_1 s_2 ...,s_n)//
   declare n, A(0:n-1),P(0:n-1),S(0:n-1),T(0:n-1)
   delcare Q(0:n-1),U(0:n-1),j,i
   for each PE(i) do in parallel
      j ← f(i,0)
      //initialize 2^0-blocks//
      S(j) ← 0; T(j) ← P(j);Q(j) ← A(j);U(j) ← A(j)+P(j)
      S(j+1) ← 0; T(j+1) ← P(j+1)
      Q(j+1) ← A(j+1);U(j+1) ← A(j+1)+P(j+1)
      for k ← 0 to ⌊log_2 n⌋ -1 do
       //combine 2^k-blocks//
       j ← f(i,k) //PE assignment//
       if j+2^k<n then
           Q(j+2^k) ← max{Q(j+2^k), U(j)+S(j+2^k)}
           U(j+2^k) ← max{U(j+2^k), U(j)+T(j+2^k)}
           U(j) ← U(j+2^k)
           S(j+2^k) ← S(j+2^k)+T(j)
           T(j+2^k) ← T(j)+T(j+2^k)
           T(j) ← T(j+2^k)
       endif
      end for
   end for
   s_i ← Q(i), 0≤i<n

end PADMIS
```

Figure 6.4: Parallel admissable schedule algorithm.

The complexity of PADMIS is readily seen to be O(logn). It uses n/2 PEs. By dividing the jobs into $\lceil n/\log n \rceil$ groups, each of size at most logn, it is possible to compute

the s_is in O(logn) time. This requires combining the sequential and parallel algorithms together. We omit the details. However, this grouping technique has been used in other problems. The details can be found in [6]. With this grouping technique, the parallel admissable schedule algorithm will have an EPU of O(1).

The overall complexity of the parallel algorithm to minimize earliness and tardiness penalties is determined by the sort (to order jobs). This takes O(logn) time and uses n^2 PEs. The EPU is O(1/n).

7. Channel Assignment

The channel assignment problem occurs naturally as a wire routing problem. Components of an electrical circuit are laied out in a straight line as in Figure 7.1. Certain pairs of components are to be connected using only two vertical runs and one horizontal run of wire (as in Figure 7.1). The horizontal and vertical runs are physically located in different layers. Each horizontal run of wire lies in a 'channel'. No channel can simultaneously carry more than one wire. We are required to assign the horizontal wire runs to channels, using the least number of channels. The assignment of Figure 7.1 uses 3 channels.

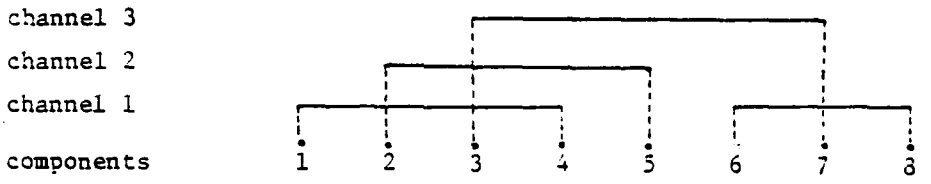


Figure 7.1: Wiring with 3 channels.

In the mathematical formulation of this problem, we are given n pairs of points (a_i,b_i), $1 \le i \le n$. Each pair (a_i,b_i) is to be joined by a continuous horizontal run of wire. These wires are to be assigned to channels, in such a way that the number of channels used is minimum. In the example of Figure 7.1, n=4; the pairs of points are (1,4), (2,5), (3,7), and (6,8); the channel assignment is: (1,4) and (6,8) in channel 1, (2,5) in channel 2, and (3,7) in channel 3.

The job sequencing problem with release times and due times [3] is similar to the channel assignment problem. Suppose we are given a set J of n jobs. Associated with each job is a release time r_i, a due time d_i, and a processing time p_i. A feasible schedule is one in which no job is

processed before its release time; all jobs complete by their respective due times; and jobs are processed without interruption from start to finish. We are required to find a feasible schedule that uses the fewest number of machines. One readily sees that when $r_i+p_i=d_i$, $1\leq i\leq n$, this problem is identical to the channel assignment problem. When this restriction on r_i, p_i, and d_i is removed, the problem is NP-hard.

The fastest sequential algorithm known for the channel assignment problem is due to Gupta, Lee, and Leung [9]. This algorithm runs in O(nlogn) time and consists of the following steps:

```
step 1: Sort the multiset {a_i | 1≤i≤n}U{b_i | 1≤i≤n}
        of the 2n end points into nondecreasing order.
step 2: m ← 0; stack ← empty
step 3: process the 2n points one by one
        if the point being processed is an a_i
          then if stack empty then m ← m+1
                  assign this wire run to channel m
               else unstack a channel from
                    the stack and assign the wire to this
                    channel
               endif
          else put the channel used by this wire onto the
               stack
        endif
```

In the above three step algorithm of [9], the final value of m is the fewest number of channels needed. The assignment is constructed while this number is being determined. It is possible to determine this number without actually obtaining a channel assignment. Let c_1, c_2, ..., c_{2n} be the sorted sequence of 2n end points. Set $z_i=1$ if c_i is an a_i and $z_i=-1$ if c_i is a b_i. It is easy to see that $r_j=\sum_{i=1}^{j} z_i$ gives the number of wires that either start at c_i or cross the point c_i. Further, $\max_{1\leq j\leq 2n}\{r_j\}$ is the number of channels needed to route the n wire segments.

r_j, $1\leq i\leq n$ can be computed using the partial sums algorithm of [6]. This algorithm takes O(logn) time and uses $\lceil n/\log n\rceil$ PEs. The largest r_j can be found in O(logn) time using $\lceil n/\log n\rceil$ PEs. The initial ordering of the as and bs can be done in O(logn) time using n^2 PEs. If this sorting algorithm is used, the resulting parallel algorithm to determine the fewest number of channels has a time complexity of O(logn) and an EPU of O(1/n). If the $O(\log^2 n)$, n PE sorting algorithm of [21] is used instead, the time complexity is $O(\log^2 n)$ and the EPU is O(1/logn).

Example 7.1: Figure 7.2 gives a set of n wires. Figure 7.3 shows the results of the different steps of the parallel algorithm to determine the fewest number of channels needed. This number is 4. []

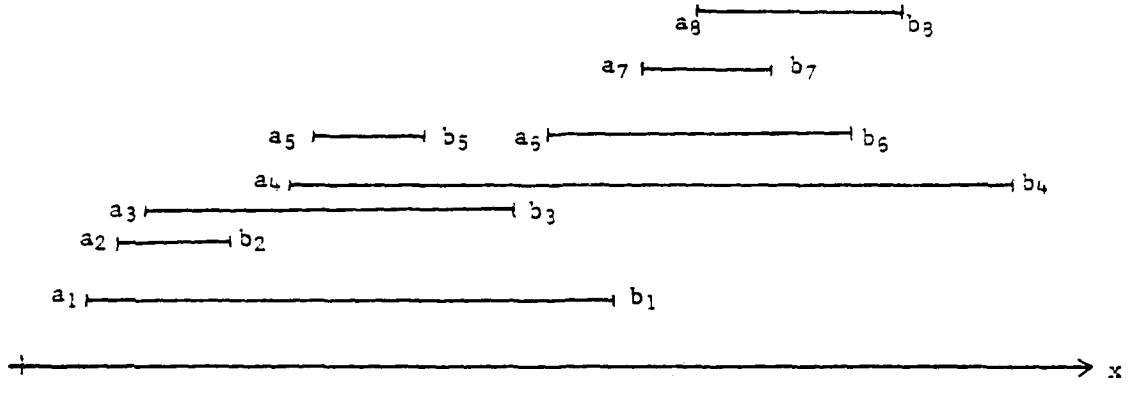


Figure 7.2

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}
Sort	a_1	a_2	a_3	b_2	a_4	a_5	b_5	b_3	a_6	b_1	a_7	a_8	b_7	a_6	b_8	b_4
Assigned Values	1	1	1	-1	1	1	-1	-1	1	-1	1	1	-1	-1	-1	-1
Partial Sum	1	2	3	2	3	4	3	2	3	2	3	4	3	2	1	0
MAX								4								

Figure 7.3

The actual channel assignment can be obtained from the r_js (recall that $r_j = \sum_{i=1}^{j} z_i$), $1 \le j \le 2n$. Assume that c_j corresponds to a_k. Let q be the largest index such that $q<j$, $r_q = r_j - 1$, and c_q corresponds to an a (say a_p). If no such q exists, set q to 0. An examination of the algorithm of Gupta et al. reveals that if $q=0$, then the channel used by (a_k, b_k) has not been used earlier. If $q \neq 0$, then it was most recently used in the interval (a_p, b_p). To see the truth of this, note that at point b_p, the channel assigned to (a_p, b_p) is put into the stack. This channel remains in the stack until we reach the nearest point at which the

number of wires that start or cross is one more than the number at b_p (if $a_i=a_j$ and $i<j$, then we say that a_i is before a_j). For every j such that c_j is an a point, let $L(k) = p$ as defined above.

L(j) partitions the set of n wires into sets. Figure 7.4 gives the paritioning for the example of Figure 5.3. Each wire is represented by a circle. The circle with index i inside it represents the wire (a_i, b_i). L(j) may be interpreted as a left link. Figure 7.4 shows the paritions as linked lists with L( ) being shown as a leftward arrow. We leave it to the reader to see how the L( ) values may be obtained in O(logn) time using $n^2/\log n$ PEs.

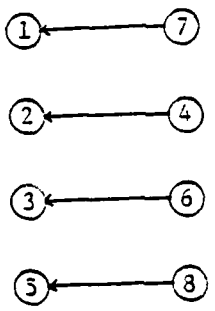


Figure 7.4: Paritions for Example 7.3

The channel assignment Q(k) for a wire k with L(k) = Ø is obtained from the r value corresponding to

If L(k) ≠ Ø, we may initially set Q(k)=Ø. The actual channel assignments for wires with L(k) ≠ Ø, may be obtained by simultaneously collapsing the linked lists and transmitting the channel assignment within the lists as below:

```
for j ← 1 to ⌈logn⌉ do
  for each i for which Q(i)=Ø do in parallel
        if L(L(i))=Ø then Q(i) ← Q(L(i))
        L(i) ← L(L(i))
  end for
end for
```

The parallel complexity of the above scheme is O(logn). Therefore, the overall complexity of our parallel channel assignment algorithm is O(logn) (i.e., using the O(logn) n^2 PE sorting algorithm); its EPU is O(1/n).

8. Conclusions

The extent to which parallel computers will find application will depend largely on our ability to find efficient algorithms for them. In this paper we have examined several scheduling problems. The single processor algorithm for each of these appeared to be highly sequential in nature. A closer look revealed a parallel structure that led to efficient parallel algorithms. Several other scheduling problems can be solved efficiently using the techniques of this paper and of [22].

Some examples are:

(a) 2 machine flow shop scheduling to minimize finish time.
(b) 2 machine open shop scheduling to minimize finish time
(c) 2 machine flow shop scheduling, with no wait in process, to minimize finish time

The parallel algorithms for the above problems involve a rather straightforward application of parallel sorting and partial sums. For example, consider problem (a). Here, we simply divide the job set into two classes: (i) jobs which need less time on machine 1 than on 2 (ii) remaining jobs. Jobs in (i) are sorted into nondecreasing order of their machine 1 processing times. Jobs in (ii) are sorted into nondecreasing order of their machine 2 processing time. The optimal processing permutation consists of jobs in (i) in sorted order followed by those in (ii) in sorted order. One readily sees that this permutation satisfies Jackson's rule [15].

References

1. Agerwala, T. and Lint, B., "Communication in Parallel Algorithms for Boolean Matrix Multiplication," Proc. 1978 Int. Conf. on Parallel Processing, IEEE pp. 146-153, 1978.
2. Arjomandi, E., "A study of parallelism in grapn theory," Ph.D. thesis, Computer Science department, University of Toronto, December 1975.
3. Batcher, K. E., "MPP - a massively parallel processor," proc. 1979 Int. Conf. on Parallel Processing, IEEE, p 249, 1979.
4. Csanky, L., "Fast parallel matrix inversion algorithms," Proc. 6th IEEE Symp on Found. of Computer Science, October 1975, pp. 11-12.
5. Dekel, E., Nassimi, D., and Sahni S., "Parallel matrix and graph algorithms," Department of Computer Science, University of Minnesota, TR 79-10, 1979, to appear in SIAM Computing.
6. Dekel, E. and Sahni, S.,"Binary Trees and papallel scheduling algorithms," Department of Computer Science, University of Minnesota, TR 80-19, 1980.
7. Eckstein, D., "Parallel graph processing using depth-first search and breadth first search," Ph.D. Thesis, University of Iowa, 1977.
8. Gertsbakh, I. and Stern, H. I., "Minimal resources for fixed and variable job schedules," Operations Research, vol. 26, No. 1, 1978, pp. 61-85.
9. Gupta, U. I., Lee, D. T., and Leung, J. Y-T., "An optimal solution for the channel-assignment problem," IEEE Trans. on Computers Vol. C-28, No. 11, 1979, pp. 807-810.
10. Higrschberg, D. S., Chandra, A. K. and Sarwate, D. V., "Computing connected components on parallel computers," CACM 22, 8(1979), pp. 461-469.
11. Hirschberg, D. S., "Fast parallel sorting algorithms," CACM, Vol. 21, No. 8, August 1978, pp. 657-661.
12. Horn, W. A., "Some simple scheduling algorithms," Naval Res. Logist. Quart., Vol. 21, pp. 177-185, 1974.
13. Horowitz, E., Sahni, S., "Exact and approximate algorithms for scheduling nonidentical processors," JACM, 23, 1976, pp. 317-327.
14. Horowitz, E. and Sahni, S., "Fundamentals of computer algorithms," Computer Science Press, Potomac, MD, 1978.
15. Jackson, J. K., "Scheduling a production line to minimize tardiness," Research report 43, Management Science Research Project, Univeristy of California, Los Angeles, 1955.
16. Lakshminarayan, S., Lakshmanan, R., Padineav, R. and Rochette, R., "Optimal single machine scheduling with earliness and tardiness penalties," Operations Research vol. 26, No. 6, 1978, pp. 1079-1082.
17. McNaughton, R., "Scheduling with deadlines and loss

functions," Management Sci. 6, 1959, pp. 1-12.
18. Moore, J. M., "An n job, one machine sequencing algorithm for minimizing the number of late jobs," Management Sci. 15, PP. 102-109, 1968.
19. Muller, D. E., and Preparata, F. P., "Bounds to complexities of networks for sorting and for switching," JACM, Vol. 22, No. 2, April 1975, pp. 195-201.
20. Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," IEEE Trans. on Computers ,c-30, No. 2, Feb 1981, PP 101-107.
21. Preparata, F. P., "New parallel-sorting schemes," IEEE Trans. on Computers, C-27, No. 7, July 1978, pp. 669-673.
22. Savage, C., "Parallel algorithms for graph theoretic problems," Ph.D. Thesis, University of Illinois, Urbana, August 1978.
23. Sidney, J. B., "Optimal single-machine scheduling with earliness and tardiness penalties," Operations Research Vol. 25, pp. 62-69, 1977.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A103	3. RECIPIENT'S CATALOG NUMBER 796
4. TITLE *(and Subtitle)* Parallel Scheduling Algorithms		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eliezer Dekel and Sartaj Sahni		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0650
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Minnesota 136 Lind Hall, 207 Church St. SE, Mpls., MN.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, Virginia 22217		12. REPORT DATE March 1981
		13. NUMBER OF PAGES 27
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Parallel algorithms, complexity, shared memory model, mean finish time, earliness, tardiness, deadline.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

We obtain fast parallel algorithms for several scheduling problems. Some of the problems considered are: scheduling to minimize the number of tardy jobs; job sequencing with deadlines; scheduling to minimize earliness and tardiness penalties; channel assignment; and minimizing the mean finish time. The shared memory model of parallel computers is used.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*