Parallel Scheduling Algorithms

ELIEZER DEKEL and SARTAJ SAHNI
University of Minnesota, Minneapolis, Minnesota
(Received March 1981; accepted April 1982)

Parallel algorithms are given for scheduling problems such as 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 to obtain fast algorithms.

WITH THE CONTINUING dramatic decline in the cost of hardware, it is becoming feasible to build economical computers with thousands of processors. In fact, Batcher [1979] 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 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 deal directly with only the single instruction stream, multiple data stream (SIMD) model. SIMD computers have the following characteristics.

1. They consist of p processing elements (PEs). The PEs 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 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 is 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 (though using possibly different data). The set of enabled PEs can change from instruction to instruction.

Subject classification: 584 parallel scheduling algorithms.

Operations Research
Vol. 31, No. 1, January–February 1983
0030-364X/83/3101-0024 $01.25

To see how an SIMD computer may be used for parallel computing, consider the problem of adding together the 8 numbers $a_0, \cdots, a_7$ on an SIMD computer with 4 PEs. This summation can be accomplished using the scheme described in Figure 1.

The number below each node in the tree of Figure 1 is a PE index. During Step 1 of the parallel algorithm, all four PEs are active (or enabled). Each executes an add instruction during this step. PEs 0, 1, 2, and 3, respectively, compute $a_0 + a_1$, $a_2 + a_3$, $a_4 + a_5$, and $a_6 + a_7$. In the next step, PEs 0 and 1 are active while PEs 2 and 3 are idle. In this step, PEs 0 and 1, respectively, compute $\Sigma_0^3 a_i$ and $\Sigma_4^7 a_i$. PE 0, for example, computes $\Sigma_0^3 a_i$ by adding together the sums computed in the first step by PEs 0 and 1. Finally, in Step 3, PE 0 uses the results computed in Step 2 by PEs 0 and 1 to obtain $\Sigma_0^7 a_i$. During this step, PEs 1, 2, and 3 are idle.

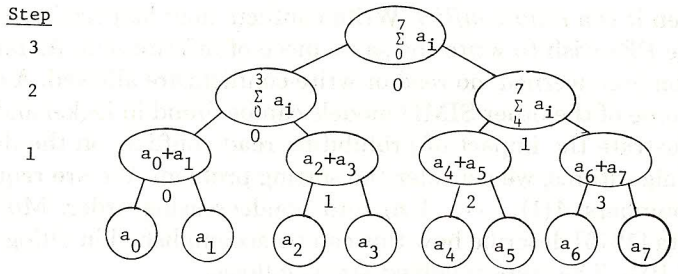

Figure 1. Parallel computation of $\Sigma_0^7 a_i$.

Note that if only 1 PE were available, then 7 steps would be needed to accomplish this summation. One should also note that the scheme just described can be generalized to add n numbers in $O(\log n)$ time using $n/2$ PEs. Actually, it is possible to add n numbers in $O(\log n)$ time using only $(n/\log n)$ PEs (see Savage [1978] or Dekel and Sahni [1981]).

Throughout this paper, we shall deal with only the SIMD model explicitly. Our techniques and algorithms readily adapt to the other models (e.g., multiple instruction stream multiple data stream (MIMD) and data flow models). This observation follows directly from the definition of these other models. An MIMD computer, for example, differs from an SIMD computer in that the PEs are asynchronous and operate under the control of individual instruction streams. This makes it possible for different PEs to execute different instructions at any given time (i.e., one PE can perform an add while another is performing a subtract, etc.). One readily sees that an SIMD algorithm can be run on an MIMD computer by simply replicating the single instruction stream over the available PEs and synchronizing at the end of each instruction.

In our earlier discussion of the parallel summation algorithm, we ignored such details as how PE 0 in Step 2 obtained the result of Step 1

from PE 1. In many SIMD models, the time required to communicate data from PE to PE often dominates the overall complexity of the algorithm. Several interprocessor communication models for SIMD computers have been proposed in the literature. Siegel [1979] summarizes some popular communication models.

The communication overhead of an algorithm varies from one communication model to another. To simplify the discussion, we deal only with the shared memory model (SMM) in this paper. This model has no communication delay. In a shared memory computer, 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 say that a *conflict* has occurred. 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 is concerned, no read or write conflicts are allowed. A description of some of the other SIMD models can be found in Dekel and Sahni.

To illustrate the impact of prohibiting read conflicts on the design of parallel algorithms, we consider the sorting problem. We are required to sort n numbers $A(1), \cdots, A(n)$ into nondecreasing order. Muller and Preparata [1975] describe how this can be accomplished in $O(\log n)$ time using n^2 PEs. The steps involved are as follows.

1. Set $C(i, p) = 1$ if either of the following is true:
   (a) $p \leq i$ and $A(p) \leq A(i)$
   (b) $p > i$ and $A(p) < A(i)$
   Set $C(i, p) = 0$ otherwise. $1 \leq i \leq n$, $1 \leq p \leq n$.
2. Compute $R(i) = \sum_{p=1}^{n} C(i, p)$, $1 \leq i \leq n$.

Note that $R(i)$ gives the position $A(i)$ is to occupy in the sorted sequence. Once $R(i)$ is known, we simply move $A(i)$ to position $R(i)$.

Performing Step 1 in parallel introduces many read conflicts. For example, $A(1)$ is needed in the computation of $C(1, p)$ and $C(p, 1)$, $1 \leq p \leq n$. To avoid read conflicts, we must first make $2n - 1$ copies of each of the $A(i)$s. Once this has been done, each of the available n^2 PEs can compute a different $C(i, p)$ using individual copies of $A(i)$ and $A(p)$.

$2n$ copies of $A(i)$ can be made in $O(\log n)$ time using n PEs as shown in Figure 2. Hence, using n^2 PEs, $2n$ copies of each of the $A(i)$s can be obtained in $O(\log n)$ time. Following this, the n^2 values $C(i, p)$, $1 \leq i \leq n$, $1 \leq p \leq n$ can be computed in $O(1)$ time. Finally, all the $R(i)$s can be obtained in parallel in $O(\log n)$ time by using $n/2$ PEs to compute each of the $R(i)$s. The overall complexity of the parallel sorting algorithm is $O(\log n)$.

Most algorithmic studies of parallel computation have been based on the SMM. Parallel matrix and graph algorithms for the SMM have been developed by Agerwala and Lint [1978], Arjomandi [1975], Csanky [1975], Eckstein [1977], Hirschberg et al. [1979], and Savage. Hirschberg [1978], Muller and Preparata, and Preparata [1978] have considered the sorting problem for SMMs. The results of Muller and Preparata, and Preparata will be used in this paper. In these two papers, it is shown that n numbers can be sorted in $O(\log n)$ time (as described above) if n^2 PEs are available, and in $O(\log^2 n)$ time when n PEs are available.

Dekel and Sahni develop a design technique for parallel algorithms based on binary computation trees. This design technique is illustrated using several examples from scheduling theory. Some of the scheduling problems considered are as follows.

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.

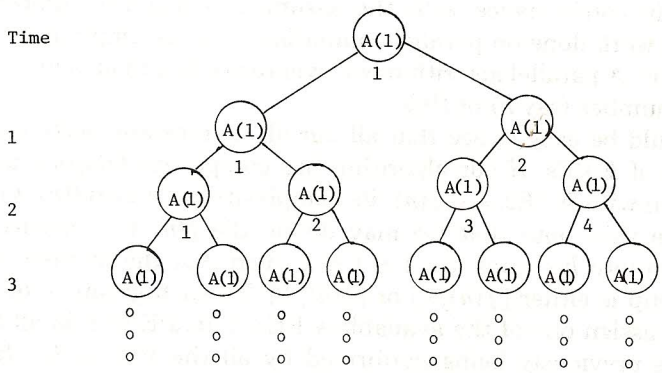

Figure 2. Replicating $A(1)$.

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

When measuring the effectiveness of a parallel algorithm, 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) = (complexity of the fastest sequential algorithm for P)/
(number of PEs used by A * complexity of A)

where an *asterisk denotes multiplication.*

The algorithm of Dekel and Sahni 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 [1974] and runs in $O(n \log n)$ time. So, the EPU of the parallel algorithm of Dekel and Sahni for P1 is $\Omega(n \log n/(n \log^2 n)) = \Omega(1/\log n)$. [The notation $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") signifies that there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$. Thus, $g(n)$ bounds $f(n)$ from below.]

The best EPU one can hope for is $\Omega(1)$. Few parallel algorithms achieve this EPU. Dekel and Sahni present some algorithms that do. One that we need here is for the partial sums problem. We are given n numbers a_1, a_2, $\cdots$, a_n and are required to compute $A_j = \oplus_{i=1}^{j} a_i$, $1 \leq j \leq n$, where $\oplus$ is any associative operator (e.g. max, min, +, *). Their algorithm runs in $O(\log n)$ time and uses $n/\log n$ 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 $O(g(n)f(n)/k)$.

To see this, note that we may divide the $f(n)$ PEs needed by the algorithm into k groups, G_i, $1 \leq i \leq k$, such that the number of PEs in each group is either $\lfloor f(n)/k \rfloor$ or $\lceil f(n)/k \rceil$. To each group G_i of required PEs, we assign one of the available k PEs. This PE will do all the work that was previously being performed by all the PEs in G_i. It will, of course, take this PE $O(f(n)/k)$ time to do the work done by the PEs in G_i in one step. Hence, with only k PEs available, the algorithm's complexity becomes $O(g(n)f(n)/k)$.

As an example, consider the summation of Figure 1. We saw that this could be done in 3 steps when 4 PEs were available. Suppose that only 2 PEs, A and B, are available. We can use PE A to do the work previously done by PEs 0 and 2, and use PE B to do the work of PEs 1 and 3. The summation would proceed as follows. In Step 1, PEs A and B would, respectively, compute $a_0 + a_1$, and $a_2 + a_3$. Next, these two PEs will compute $a_4 + a_5$, and $a_6 + a_7$. In Step 3, $\Sigma_0^3 a_i$ and $\Sigma_4^7 a_i$ will be computed. Finally, in Step 4, $\Sigma_0^7 a_i$ will be computed. Note that in this case, computing

with 4 PEs is not twice as fast as with 2. This is because the 4 PE algorithm used all 4 PEs in Step 1 only.

We continue with tradition, and explicitly analyze our algorithms only for the case when as many PEs as needed are available.

In Sections 1 and 2, 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 3, 4, 5, and 6, we respectively, consider the following problems: (i) minimizing the number of tardy jobs when $p_i = 1$, $1 \leq i \leq n$ and 1 machine is available, (ii) job sequencing with deadlines, (iii) scheduling one machine to minimize the maximum earliness and tardiness penalties, and (iv) channel assignment.

1. MINIMUM FINISH TIME

When preemptions are permitted, a minimum finish time schedule for m machines is efficiently obtained using McNaughton's rule [1959]. Let $p_1, p_2, \cdots, p_n$ be the processing times of the n jobs. The finish time, f, of an optimal preemptive schedule is given by:

$$f = \max\{\max_{1\leq i\leq n}\{p_i\}, (1/m)\Sigma_{i=1}^{n} p_i\}.$$

Using f, the optimal schedule may be constructed in $O(n)$ time. 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 0. 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 Dekel and Sahni, $\max\{p_i\}$ and $\Sigma_{j=1}^{n} p_i$, may be computed in $O(\log n)$ time with $n/\log n$ PEs. To obtain the actual schedule, we also need $A_i = \Sigma_{j=1}^{n} p_j$, $1 \leq i \leq n$. As mentioned in the introduction, all the A_is can be computed in $O(\log n)$ time using $n/\log n$) PEs. Let $A_0 = 0$. Each job i can now determine its own processing assignment by using the following rule:

case
$x \leftarrow \lceil A_{i-1}/f \rceil * f - A_{i-1}$
  :$x = 0$: schedule job i on machine $\lceil A_i/f \rceil$ from 0 to p_i
  :$x \geq p_i$: schedule job i on machine $\lceil A_i/f \rceil$ from $f - x$ to $f - x + p_i$
  :else: schedule job i on machine $\lceil A_i/f \rceil$ from 0 to $p_i - x$ and on machine $\lceil A_i/f \rceil - 1$ from $f - x$ to f.
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.

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 3

Example 1. Suppose we have 14 jobs with processing times as given in Figure 3. Let $m = 5$. $f = max\{7, 50/5\} = 10$. Figure 3 gives the A_i and x values for each job. The schedule obtained is given in Figure 4.

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

2. MINIMUM MEAN FINISH TIME

A nonpreemptive schedule that minimizes the mean finish time of n jobs on m identical machines is obtained by using the SPT 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 [1976] present an $O(n \log mn)$ algorithm that constructs a minimum mean finish time schedule for this case. Their algorithm is reproduced in Figure 5. This algorithm assumes that the speeds and processing times have been normalized and sorted such that $s_i = 1 \leq s_2 \leq \cdots \leq s_n$ 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

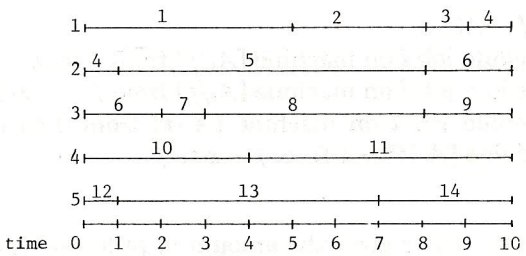

Figure 4

nondecreasing order. Let the resulting sequence be $a_1, a_2, a_3, \cdots, a_{mn}$. 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 short and mn PEs or in $O(\log mn)$ time with m^2n^2 PEs. If we use the former sort algorithm, the EPU of our parallel algorithm is $\Omega(n \log mn/(mn \log^2 mn)) = \Omega(1/(m \log mn))$. If the latter sort algorithm is used, the EPU of our scheduling algorithm becomes $\Omega(n \log mn/(m^2n^2 \log mn)) = \Omega(1/(m^2n))$. The actual start and finish times for each job can be obtained by later using the partial sums algorithm of Dekel and Sahni.

Algorithm MFT

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

Output: Sets R_i, $1 \leq i \leq m$. 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 $i_u = \min_{1 \leq j \leq m}\{i_j\}$; $R_u \leftarrow R_u \cup \{k\}$; $i_u \leftarrow i_u + 1/s_u$
end for
end MFT

Figure 5

3. NUMBER OF TARDY JOBS

Let $J = \{(p_i, d_i) | 1 \leq i \leq n\}$ 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 (see Moore [1968]) have developed an $O(n \log n)$ sequential algorithm that obtains a schedule that minimizes the number of tardy jobs. Dekel and Sahni 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 $\Omega(1/\log n)$.

In this section, we develop a parallel algorithm for the case when $p_i = 1$, $1 \leq i \leq n$. This algorithm will have a complexity $O(\log n)$. It will require $O(n^2)$ PEs and thus have an EPU that is $\Omega(1/n)$. While the algorithm of this section has an EPU that is inferior to that of Dekel and

Sahni, 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. In a recent paper, Monma [1982] has presented an $O(n)$ solution to this problem.

The problem of finding a schedule that minimizes the number of tardy jobs is equivalent to 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 (for example see Horowitz and Sahni 1978]).

When $p_i = 1$, $1 \le i \le n$, a minimum 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 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 $O(n \log n)$ as it takes this much time to order the jobs by due time.

Let J be a set of n unit processing time jobs. Let $D(i)$, $1 \le i \le k$ be the distinct due times of the jobs in J. Assume that $D(i) < D(i + 1)$, $1 \le i \le k$. Let $n(i)$ be the number of jobs in J with due time $D(i)$, $1 \le i \le k$. Clearly, $\Sigma_{i=1}^{k} n(i) = n$. Let $D(0) = 0$ and $n(0) = 0$. Define $F(i)$ to be the value of j when procedure FEAS (Figure 6) has just finished considering all jobs in J with due time at most $D(i)$. It is evident that:

$$\begin{aligned} &F(0) = D(0) = 0 \\ &F(i) = \min\{F(i-1) + n(i), D(i), b\}, \qquad 1 \le i \le k. \end{aligned} \tag{1}$$

Expanding the recurrence (1), we obtain:

$$\begin{aligned} F(1) &= \min\{D(0) + n(1), D(1), b\} \\ F(2) &= \min\{F(1) + n(2), D(2)\ b\} \\ &= \min\{D(0) + n(1) + n(2), D(1) + n(2), b + n(2), D(2), b\} \\ &= \min\{D(0) + n(1) + n(2), D(1) + n(2), D(2), b\} \\ F(3) &= \min\{D(0) + n(1) + n(2) + n(3), D(1) + n(2) + n(3), \\ &\quad D(2) + n(3), D(3), b\} \\ &\vdots \end{aligned}$$

And, in general

$$F(m) = \min\{\min_{0 \le i \le m}\{D(i) + \Sigma_{q=i+1}^{m} n(q)\}, b\}, \qquad 1 \le m \le k. \tag{2}$$

```
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 6

The maximum number of jobs in J that can be scheduled in $[0, b]$, $b > 0$, so that none are 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 $d(1), d(2), \cdots, d(n)$. Let $d(0) = 0$. We may assume that $d(i) > 0$, $1 \leq i \leq n$. The computation steps are:

Step 1. Sort $d(1{:}n)$ into nondecreasing order.

Step 2. Determine the positions (or points) $r(0), \cdots, r(k-1)$ in the sorted sequence of due times in $d(0{:}n)$ where the due times changes. I.e., $r(i) < r(i+1)$, $1 \leq i < k$ and $d(r(i)) \neq d(r(i)+1)$. Let $r(k) = n$. Clearly, $r(0) = 0$, and $n(i) = r(i) - r(i-1)$ and $D(i) = d(r(i))$, $1 < i < k$; $D(0) = 0$.

Step 3. Since $D(i) + \Sigma_{q=i+1}^{k} n(q) = D(i) + n - r(i)$, we compute $F(k) = \min\{n + \min_{0 \leq i \leq k}\{D(i) - r(i)\}, b\}$.

Example 2. Figure 7a gives the due times of a set J of 15 jobs. In Figure 7b, have been ordered by due times. The points at which the due

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 7

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, $n + \min_{0 \le i \le k}\{D(i) - r(i)\} = 15 + \min\{0, -1, -4, -3, -1, -4, -4\} = 15 - 4 = 11$. 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 Muller and Preparata, and Preparata). 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

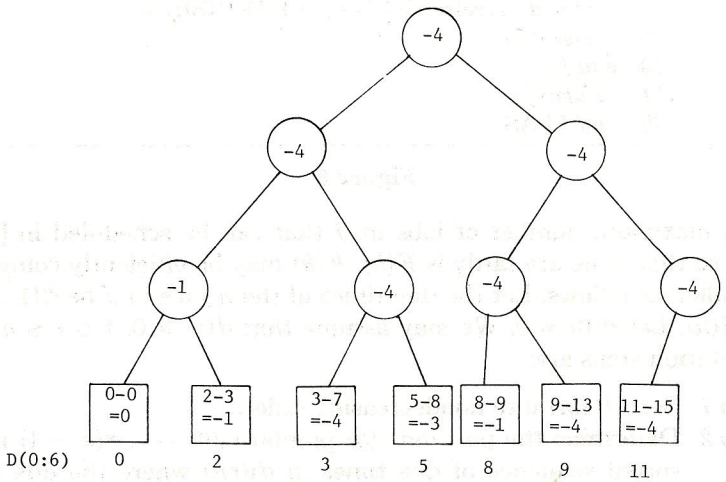

Figure 8

$r(0), r(1), \cdots, r(k)$. This can be done in $O(\log n)$ time using n PEs and the data concentration algorithm of Nassimi and Sahni [1981]. Another data concentration step moves $d(r(0)), d(r(1)), \cdots, d(r(k))$ into $D(0), D(1), \cdots, D(k)$. 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 Dekel and Sahni. (Figure 8 shows this for our example.) As explained in Dekel and Sahni, only $O(k/\log k)$ PEs are needed for this; but using $k/2$ PEs is 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 $\Omega(n \log n/(\log n{*}n^2)) = \Omega(1/n)$.

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) $\leftarrow j$

and by deleting lines 3 and 7.

It is easy to see that job i is completed at time DONE(i) iff DONE($i-1$) $\neq$ DONE(i), $1 \leq i \leq n$. For the modified algorithm, we see that:

$$\text{DONE}(0) = 0$$

$$\text{DONE}(i) = \min\{\text{DONE}(i-1) + 1, d_i\}, \qquad 1 \leq i \leq n. \quad (3)$$

Solving (3), we obtain:

$$\text{DONE}(i) = \min_{0 \leq j \leq i}\{d_j + i - j\}, \qquad 1 \leq i \leq n \quad (4)$$

DONE(i), $1 \leq i \leq n$ may be computed in $O(\log n)$ time using n^2 PEs (though $n^2/\log n$ PEs are sufficient) and the binary computation tree model (see Dekel and Sahni and Figure 8). Since the initial sort takes $O(\log n)$ time and requires n^2 PEs, the overall time complexity is $O(\log n)$ and the EPU is $\Omega(1/n)$. From DONE(i), the schedule is easily obtained.

Example 3. For the sorted data of Example 2, 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 nontardy jobs 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.

4. 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, $1 \leq i \leq n$. 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 shown in Figure 9. A correctness proof of this procedure may be found in Horowitz and Sahni [1978]. It is also possible to implement this scheme by a sequential algorithm of complexity $O(n \log n)$. 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 1. *Let A be a feasible subset of J that yields maximum return. Let B be any feasible subset of J. $|B| \leq |A|$.*

Step 1. Sort J into nonincreasing order of z_i
Step 2. $R \leftarrow \{1\}$
for $i \leftarrow 2$ *to* n *do*
if $R \cup \{i\}$ is feasible *then* $R \leftarrow R \cup \{i\}$
end for

Figure 9

Proof. Since A and B are feasible subsets, they can respectively be scheduled in $[0, |A|]$ and $[0, |B|]$ 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 10(a)). A similar transformation may be made if i is scheduled later in SA than in SB (see Figure 10(b)).

By performing the above transformation on all jobs that are in both A and B (i.e., in the intersection of A and B), we obtain schedules SA' and SB' that contain no tardy jobs. In addition, jobs that are in both A and 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 is not in 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 1, we may derive a parallel algorithm. Let $T1(i) = \{j | z_j > z_i$ or $(z_j = z_i$ 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 nontardy 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 9) 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 3. From R, the optimal 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

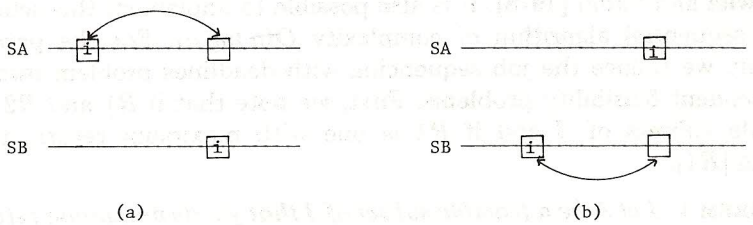

Figure 10. Lining up common jobs.

be carried out by first concentrating the jobs in R and then sorting them by due times.

Example 4. Figure 11a shows an example job set with 12 jobs. These have been ordered by due times in Figure 11b. Figure 12 gives $T2(i)$, $1 \leq i \leq n$. The number of nontardy 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 13. This gives the optimal schedule.

Complexity Analysis

As far as the complexity is concerned, the initial sort by due times can be done in $O(\log n)$ time using n^2 PEs. Next, we need to replicate this

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	85	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 11

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 $O(\log n)$ time (the $O(\log n)$ 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 $O(\log n)$ 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 $x(i) = F(k_i)$, $1 \leq i \leq n$. This takes $O(\log n)$ time. $y(i)$, $1 \leq i \leq n$ 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 $y(i) > x(i)$, $1 \leq i \leq n$. The selected jobs can be concentrated in $O(\log n)$ 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 $O(\log n)$. It uses n^2 PEs and has an EPU of $\Omega(1/n)$. This should be contrasted with the algorithm presented by Dekel and Sahni for the same problem. That

$T2(1)$	i	1
$z_1 = 85$	d_i	2
		include (0, 1)
$T2(2)$	i	1 4 6 10 2 3 5 7 8 9 11 12
$z_2 = 55$	d_i	2 2 2 3 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 (6, 7)
$T2(4)$	i	1 4 6 11
$z_4 = 80$	d_i	2 2 2 7
		reject (3, 3)
$T2(5)$	i	1 4 6 5 8 9 11
$z_5 = 70$	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
$z_7 = 60$	d_i	2 2 2 6 6 6 6 7 7
		reject (7, 7)
$T2(8)$	i	1 4 6 8 11
$z_8 = 80$	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)
$T2(10)$	i	1 4 6 10 3 5 7 8 9 11
$z_{10} = 60$	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 10 3 5 7 8 9 11 12
$z_{12} = 60$	d_i	2 2 2 3 6 6 6 6 7 7 8
		include (7, 8)

Figure 12

	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 13. The optimal schedule.

algorithm has a complexity of $O(\log^2 n)$ but uses only $O(n)$ PEs. Thus, its EPU is $\Omega(1/\log n)$.

5. 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 $(s_1, s_2, \cdots, s_n)$ where s_i is the start time of job i. Schedule S is *admissable* iff $s_i \geq s_{i-1} + p_{i-1}$, $2 \leq i \leq n$. The *completion 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., $e_i > 0$) then it incurs a penalty $g(e_i)$. If it is tardy (i.e., $t_i > 0$), 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 [1977]. 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 $g(0) = h(0) = 0$.

Our notations and definitions are taken from Sidney's paper. Sidney's $O(n^2)$ algorithm was subsequently improved to $O(n \log n)$ by Lakshminarayan et al. [1978]. The parallel algorithm developed here is based on the improved algorithm.

The algorithm of Lakshminarayan et al. first finds an admissable schedule S using procedure ADMIS (Figure 14). 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 computed next. If $\Delta = 0$, then S is clearly optimal (as $\max\{e_i\} = \max\{t_i\} = 0$). If $\Delta > 0$, 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.

Δ can be computed in $O(\log n)$ time using n PEs (see Dekel and Sahni). As described in Lakshminarayan et al., E^* may be computed in $O(1)$ time using 1 PE. Once E^* has been obtained, n copies of it can be made in $O(\log n)$ time using n PEs. Finally, the s_i's can be updated in $O(1)$ time using n PEs. Also, the initial ordering of the jobs can be carried out in $O(\log n)$ time with n^2 PEs. All that remains is the computation of the admissable schedule. From Figure 14, we see that

$$\begin{aligned} s_1 &= a_1 \\ s_i &= \max\{a_i, s_{i-1} + p_{i-1}\}, \qquad 2 \leq i \leq n. \end{aligned} \tag{5}$$

Expanding the recurrence (5), we obtain

$$s_i = \max_{1 \le j \le i}\{a_j + \Sigma_{k=j}^{i-1} p_k\}, \qquad 1 \le i \le n. \quad (6)$$

It should be easy to see that using (6) and $O(n^3)$ PEs, one can compute all the s_i's in $O(\log n)$ time. We devote the rest of this section to the development of an $O(\log n)$ algorithm that utilizes only $n/2$ PEs. As we see later, $\lceil n/\log n \rceil$ PEs are all that is needed.

For convenience, we assume that the jobs are indexed $0, 1, \cdots, n-1$ rather than $1, 2, \cdots, n$. 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-

```
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 14

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 $[i, i+1, i+2, \cdots, r]$ 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 $S(j) = \Sigma_{q=i}^{j-1} P(q)$, j is an index in this block

$$T(j) = \Sigma_{q=i}^{r} P(q), \quad (7)$$

j is a block index and r is the highest index in the block, $Q(j) = \max_{i \le q \le j}\{A(q) + \Sigma_{t=q}^{j-1} P(t)\}$ and $U(j) = Q(r) + P(r)$, where j is a block index. For a 2^0-block $[i]$, we have:

$$S(i) = 0; \quad T(i) = P(i); \quad Q(i) = A(i); \quad U(i) = A(i) + P(i) \quad (8)$$

Let $X = [i, i+1, \cdots, u]$ and $Y = [u+1, \cdots, v]$ be two sibling 2^k-blocks. Their union $Z = [i, i+1, \cdots, v]$ is a 2^{k+1}-block. Let S, T, Q, and U be the values defined in (7) 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 (7), we see that:

$$S'(j) = \begin{cases} S(j) & \text{if } j \in X \\ S(j) + T(i) & \text{if } j \in Y \end{cases} \tag{9a}$$

$$T'(j) = \begin{cases} T(j) + T(u+1) & \text{if } j \in X \\ T(j) + T(i) & \text{if } j \in Y \end{cases} \tag{9b}$$

$$Q'(j) = \begin{cases} Q(j) & \text{if } j \in X \\ \max\{Q(j),\ U(i) + S(j)\} & \text{if } j \in Y \end{cases} \tag{9c}$$

$$U'(j) = Q'(v) + P(v). \tag{9d}$$

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

$$Q(j) = \max_{0 \le q \le j}\{a_q + \Sigma_{t=q}^{j-1}\, p_t\} = s_j \quad \text{of (6)}.$$

Our strategy is to compute the admissable schedule obtained by procedure ADMIS by using (9a–d). We start with the S, T, Q, and U

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 15. An example data set.

values for 2^0-blocks as given by (8). Next using (9a–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 5. Figure 15 gives a set of 10 jobs (indexed 0 through 9). The first row of Figure 16 gives the S, T, Q, and U values for the 2^1-blocks; etc. The numbers with arrows give PE assignments. From the bottom-most row, we obtain $S = (0, 3, 5, 8, 12, 13, 16, 20, 21, 24)$ as the admissable schedule.

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 $0, 1, \cdots, \lfloor n/2 \rfloor - 1$. With respect to our example of Figure 16, 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 are both assigned to the new 2^2-block $[0, 1, 2, 3]$, being constructed. PEs 2 and 3 are assigned to the block $[4, 5, 6, 7]$. PE 4 is idle.

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 $f(i, j)$ such that the binary representation of the value of $f(i, j)$ is $\cdots i_{j+1}, i_j, 0, i_{j-1} \cdots i_0$. 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 17. This algorithm mirrors Equations 9a–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 (9a) has been replaced by $S(j)$

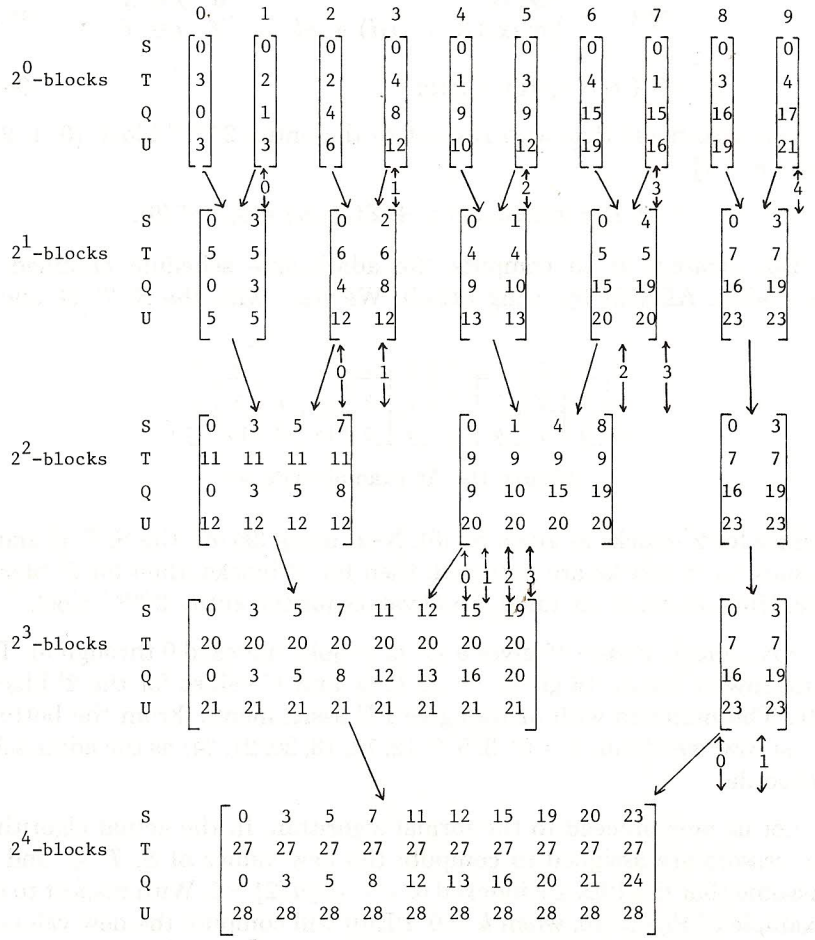

Figure 16. Computing the admissable schedule.

$+ T(j - 2^k)$. Similarly, $T(j) + T(u + 1)$ of (9b) has been replaced by $T(j) + T(j + 2^k)$; and $U(i) + S(j)$ of (9c) 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 be 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. 9d). When $k = \lfloor \log n \rfloor - 1$, only Q need be computed.

The complexity of PADMIS is readily seen to be $O(\log n)$. It uses $n/2$ PEs. By dividing the jobs into $\lceil n/\log n \rceil$ groups, each of size at most $\log n$, it is possible to compute the s_i's in $O(\log n)$ time. This requires combining the sequential and parallel algorithms together. We omit the details.

```
Procedure PADMIS (A, P, s, n)
  //obtain the admissable schedule (s1, s2, ..., sn)//
  declare n, A(0:n − 1), P(0:n − 1), S(0:n − 1), T(0:n − 1)
  declare 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 ⌊log2 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
  si ← Q(i), 0 ≤ i < n
end PADMIS
```

Figure 17. Parallel admissable schedule algorithm.

However, this grouping technique has been used in other problems. The details can be found in Dekel and Sahni. With this grouping technique, the parallel admissable schedule algorithm (Figure 17) will have an EPU of $\Omega(1)$.

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

6. CHANNEL ASSIGNMENT

The channel assignment problem occurs naturally as a wire routing problem. Components of an electrical circuit are laid out in a straight line

as in Figure 18. Certain pairs of components are to be connected using only two vertical runs and one horizontal run of wire (as in Figure 18). 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 18 uses 3 channels.

In the mathematical formulation of this problem, we are given n open intervals (a_i, b_i), $a_i < b_i$, $1 \leq i \leq n$. Each open interval (a_i, b_i) corresponds to a continuous horizontal run of wire that joins a pair of components. 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 18, $n = 4$; the intervals 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.

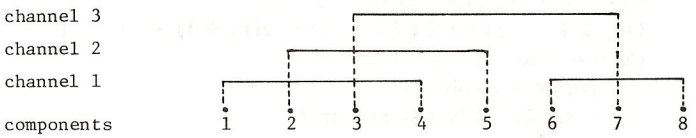

Figure 18. Wiring with 3 channels.

The job sequencing problem with release times and deadlines (Gertsbakh and Stern [1978]) 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 deadline 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 deadlines; 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 runs in $O(n \log n)$ time (Hashimoto and Stevens [1971], Kernighan et al. [1973], and Gupta et al. [1979]). The algorithm described by Gupta et al. consists of the steps given in Figure 19.

In the three-step algorithm of Gupta et al., 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 $c_1, c_2, \cdots, c_{2n}$ be the sorted sequence of $2n$ endpoints. Set $z_i = 1$ if c_i is an a_j and $z_i = -1$ if c_i is a b_k. It is easy to see that $r_j = \Sigma_{i=1}^{j} z_i$ gives the number of wires

Step 1. Sort the multiset $\{a_i | 1 \le i \le n\} \cup \{b_i | 1 \le i \le n\}$ of the $2n$ end points into nondecreasing order. If an a_i equals a b_j, then b_j precedes a_i.
Step 2. $m \leftarrow 0$; stack $\leftarrow$ empty
Step 3. Process the $2n$ points one by one
if the point being processed is an a_i
then if stack empty *then* $m \leftarrow m + 1$ 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

Figure 19

that either start at c_j or cross the point c_j. Further, $\max_{1 \le j \le 2n}\{r_j\}$ is the number of channels needed to route the n wire segments.

r_j, $1 \le i \le n$ can be computed using the partial sums algorithm of Dekel and Sahni. This algorithm takes $O(\log n)$ time and uses $\lceil n/\log n \rceil$ PEs. The largest r_j can be found in $O(\log n)$ time using $\lceil n/\log n \rceil$ PEs. The initial ordering of the a's and b's can be done in $O(\log n)$ 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 $O(\log n)$ and an EPU of $\Omega(1/n)$. If the $O(\log^2 n)$, n PE sorting algorithm of Preparata is used instead, the time complexity is $O(\log^2 n)$ and the EPU is $\Omega(1/\log n)$.

Example 6. Figure 20 gives a set of n wires. Figure 21 shows the results of the different steps of the parallel algorithm to determine the fewest number of channels needed. This number is 4.

The actual channel assignment can be obtained from the r_j's (recall that $r_j = \Sigma_{i=1}^{j} z_i$), $1 \le j \le 2n$. 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 a b (say b_p). If no such q exists, set q to 0. An examination of the algorithm

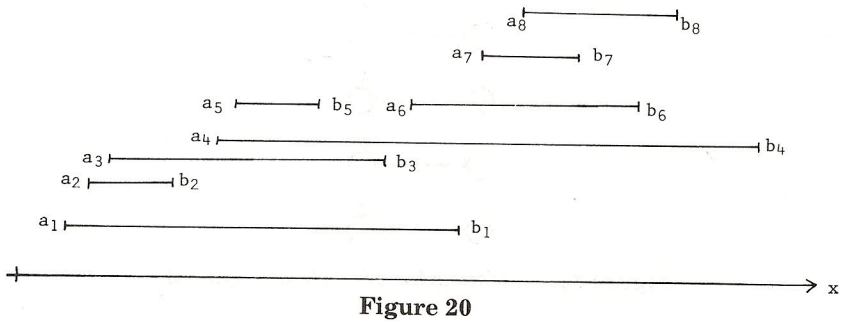

Figure 20

	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	b_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 21

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). Define $L(j) = 0$ if $q = 0$ and $L(j) = p$ otherwise.

$L(j)$ may be interpreted as a link. The value $L(j)$ links the jth wire to the previous wire which uses the same channel. Thus the $L(\ )$ values can be used to create a linked list of wires that are assigned to the same channel. Figure 22 gives the linked lists for the example of Figure 21. Each wire is represented by a circle. The circle with index i outside it represents the wire (a_i, b_i). $L(\ )$ is shown as a leftward arrow. We leave it to the reader to see how the $L(\ )$ values may be obtained in $O(\log n)$ time using $n^2/\log n$ PEs.

The channel assignment $Q(k)$ for a wire k with $L(k) = 0$ is obtained from the r value corresponding to it (i.e., $\Sigma_{i=1}^{k} z_i$).

If $L(k) \neq 0$, we may initially set $Q(k) = 0$. The actual channel assignments for wires with $L(k) \neq 0$, may be obtained by simultaneously collapsing the linked lists and transmitting the channel assignment within the lists as below:

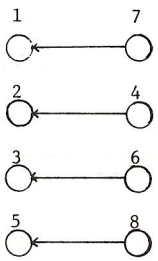

Figure 22. Partitions for example of Figure 21.

for $j \leftarrow 1$ *to* $\lceil \log n \rceil$ *do*
  for each i for which $Q(i) = 0$ *do in parallel*
    if $L(L(i)) = 0$ *then* $Q(i) \leftarrow Q(L(i))$
    $L(i) \leftarrow L(L(i))$
  end for
end for.

As an example, consider the partition shown in Figure 23a. The letters outside the circles identify the wires. The numbers inside give the Q

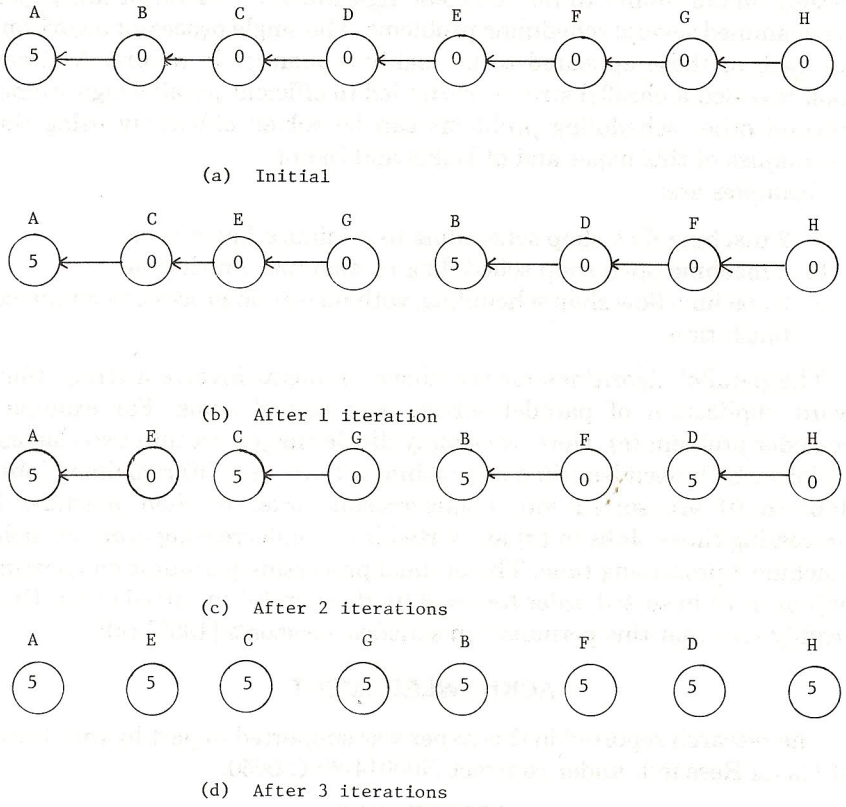

Figure 23

values. Thus, initially the channel assignment for wire A is known to be 5. The remaining wires B–H do not know their channel assignments. It is also known that wires A–H are to be assigned to the same channel. In the first iteration of the outermost *for* loop above, wires B–H look two nodes to the left and update their left links. Wire B also updates its Q value. The result is shown in Figure 23b. The results of the second and

third iterations are shown in Figure 23, c and d, respectively. Performing any additional iterations has no effect on the Q and L values.

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

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 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 Dekel and Sahni.

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 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 [1955] rule.

ACKNOWLEDGMENT

The research reported in this paper was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

REFERENCES

AGERWALA, T., AND B. LINT. 1978. Communication in Parallel Algorithms for Boolean Matrix Multiplication. *Proc. 1978 Int. Conf. on Parallel Processing*, IEEE, pp. 146–153.

ARJOMANDI, E. 1975. A Study of Parallelism in Graph Theory, Ph.D. thesis, Computer Science Department, University of Toronto.

BATCHER, K. E. 1979. MPP—A Massively Parallel Processor. *Proc. 1979 Int. Conf. on Parallel Processing*, IEEE, p. 249.

CSANKY, L. 1975. Fast Parallel Matrix Inversion Algorithms. *Proc. 6th IEEE Symp. on Found. of Computer Science*, pp. 11–12.

DEKEL, E., AND S. SAHNI. 1981. Binary Trees and Parallel Scheduling Algorithms. *Proc. CONPAR 81,* pp. 480–492. Springer-Verlag, New York.

ECKSTEIN, D. 1977. Parallel Graph Processing Using Depth-First Search and Breadth First Search, Ph.D. thesis, University of Iowa.

GERTSBAKH, I., AND H. I. STERN. 1978. Minimal Resources for Fixed and Variable Job Schedules. *Opns. Res.* **26,** 61–85.

GUPTA, U. I., D. T. LEE AND J. Y. LEUNG. 1979. An Optimal Solution for the Channel-Assignment Problem. *IEEE Trans. Computers* **C-28,** 807–810.

HASHIMOTO, A., AND J. E. STEVENS. 1971. Wire Routing by Optimizing Channel Assignment within Large Apertures. *Proc. 8th Design Automation Conference,* IEEE, pp. 155–169.

HIRSCHBERG, D. S., A. K. CHANDRA AND D. V. SARWATE. 1979. Computing Connected Components on Parallel Computers. *Commun. A.C.M.* **22,** 461–469.

HIRSCHBERG, D. S. 1978. Fast Parallel Sorting Algorithms. *Commun. A.C.M.* **21,** 657–661.

HORN, W. A. 1974. Some Simple Scheduling Algorithms. *Naval Res. Logist. Quart.* **21,** 177–185.

HOROWITZ, E., AND S. SAHNI. 1976. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *J. Assoc. Comput. Mach.* **23,** 317–327.

HOROWITZ, E., AND S. SAHNI. 1978. *Fundamentals of Computer Algorithms.* Computer Science Press, Potomac, Md.

JACKSON, J. K. 1955. Scheduling a Production Line to Minimize Tardiness, Research Report 43, Management Science Research Project, University of California, Los Angeles.

KERNIGHAN, B. W., D. G. SCHWEIKERT AND G. PERSKY. 1973. An Optimum Routing Algorithm for Polycell Layouts of Integrated Circuits. *Proc. 10th Design Automation Conference,* IEEE, pp. 50–59.

LAKSHMINARAYAN, S., R. LAKSHMANAN, R. PADINEAV AND R. ROCHETTE. 1978. Optimal Single Machine Scheduling with Earliness and Tardiness Penalties. *Opns. Res.* **26,** 1079–1082.

MCNAUGHTON, R. 1959. Scheduling with Deadlines and Loss Functions. *Mgmt. Sci.* **6,** 1–12.

MOORE, J. M. 1968. An *n* job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Mgmt. Sci.* **15,** 102–109.

MONMA, C. L. 1982. Linear Time Algorithms for Scheduling on Parallel Processors. *Opns. Res.* **30,** 116–124.

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.

NASSIMI, D., AND S. SAHNI. 1981. Data Broadcasting in SIMD Computers. *IEEE Trans. Comput.* **C-30,** 101–107.

PREPARATA, F. P. 1978. New Parallel-Sorting Schemes. *IEEE Trans. Comput.* **C-27,** 669–673.

SAVAGE, C. 1978. Parallel Algorithms for Graph Theoretic Problems, Ph.D. thesis, University of Illinois, Urbana.

SIEGEL, H. J. 1979. A Model of SIMD Machines and a Comparison of Various Interconnection Networks. *IEEE Trans. Comput.* **C-28,** 907–917.

SYDNEY, J. B. 1977. Optimal Single-Machine Scheduling with Earliness and Tardiness Penalties. *Opns. Res.* **25,** 62–69.