<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

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

Parallel Scheduling Algorithms,

by Eliezer Dekel and Sartaj Sahni
Technical Report 81-1

March 1981

Cover design courtesy of Ruth and Jay Leavitt

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited

81 7 20 039
Parallel Scheduling Algorithms
Eliezer Dekel and Sartaj Sanni
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 (e.g., multiple instruction stream multiple data stream (MIMD) and data flow models). SIMD computers have the following characteristics:

1. They consist of \( p \) processing elements (PEs). The PEs are indexed \( 0, 1, \ldots, p-1 \) and an individual PE may be referenced as in \( \text{PE}(i) \). 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 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 here is concerned, no read or write conflicts are allowed.
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], Arjomand [2], Cerny [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 PEs are available, and in O(log n) time when n PEs are available.

Dekel and Sanni [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 \( p_j = 1 \).

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 \( O(\log n) \).

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:

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

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 $a_1, a_2, ..., a_n$ and are required to compute $A_j = \oplus a_i$, $1 \leq j \leq n$, 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 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 $p_i = 1$ for $1 \leq i \leq n$ 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 McNaughton's rule [17]. Let $p_1, p_2, ..., p_n$ be the processing times of the $n$ jobs. The finish time, $f$, of an optimal preemptive schedule is given by:

$$f = \max \left( \max \left\{ \frac{1}{m} \sum_{i=1}^{n} p_i \right\} \right)$$

Using $f$, the optimal schedule may be constructed in $O(n)$ time [17]. Job 1 is scheduled on machine 1 from $j$ 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 $J$. 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 $O(\log n)$ time with $\frac{n}{\log n}$ 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 $A_i$s can be computed in $O(\log n)$ time using $n/\log n$ PEs ([6]). Let $A_0 = 0$. Each job $i$ can now determine its own processing assignment by using the following rule:

$$x \leftarrow \left\lceil \frac{A_{i-1}}{f} \right\rceil \times f - A_{i-1}$$

**case**

$x = 0$: schedule job $i$ on machine $\left\lceil \frac{A_i}{f} \right\rceil$ from 0 to $p_i$

$x > p_i$: schedule job $i$ on machine $\left\lceil \frac{A_i}{f} \right\rceil$ from $f-x$ to $f-x+p_i$

$\text{else}$: schedule job $i$ on machine $\left\lceil \frac{A_i}{f} \right\rceil$ from 0 to $p_i-x$

**end case**

One may verify that $x$ gives the amount of processing time left on the machine $\left\lceil \frac{A_{i-1}}{f} \right\rceil$ 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.

<table>
<thead>
<tr>
<th>Job</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>$A_i$</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>18</td>
<td>22</td>
<td>23</td>
<td>28</td>
<td>30</td>
<td>34</td>
<td>40</td>
<td>41</td>
<td>47</td>
<td>50</td>
</tr>
<tr>
<td>$x$</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 2.1**

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 EPJ is $O(n/(\log n \cdot n \cdot \log n)) = O(1)$. 
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 $i$, $p_j/s_i$ 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 $s_1 < s_2 < \ldots < s_m$ and $p_1 < p_2 < \ldots < p_n$.

By examining this algorithm, we see that another way to obtain an optimal schedule is to sort the $mn$ numbers $i/s_i$, $1 \leq i \leq m$, $1 \leq j \leq n$, into nondecreasing order. Let the resulting sequence be $a_1, a_2, a_3, \ldots, a_{mn}$: If $a_i$ corresponds to $q/s_q$, then job $A_{i+q}$ 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 $m^2$ PEs. If we use the former sort algorithm, the EPU of our parallel algorithm is $O(n \log mn/(mn \log mn)) = O(1/(m \log mn))$. If the latter sort algorithm is used, the EPU of our scheduling algorithm becomes $O(n \log mn/(m^2 n \log mn)) = O(1/(m^2 n))$. The actual start and finish times for each job can be obtained by later using the partial sums algorithm of [8].
Algorithm MFT

Input: $m$ processors with speeds $s_1, s_2, \ldots, s_m$.
$n$ jobs initially sorted so that $p_1 < p_2 < \ldots < p_n$, where the times $p_i$ are for processor $i$.

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

for $j \leftarrow 1$ to $m - 1$ do
    $R_i \leftarrow \emptyset$;
    $i_j \leftarrow 1/s_j$
end for

$R_m \leftarrow \{n\}$; $i \leftarrow 2/s_m$

// 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 $i_u = \min \{i_j\}$
    $i \leftarrow i_u + 1/s_u$
end for

end MFT

Figure 3.1

4. 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 [18] have developed an $O(n \log n)$ sequential algorithm that obtains a schedule that minimizes the number of tardy jobs. Dkel and Sahni [8] 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 $p_i = 1$, $1 \leq i \leq n$. This algorithm will have a complexity $O(\log n)$. It will require $O(n^2)$ PEs and thus have an EPU that is $O(1/n)$. 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 $p_i = 1$, $1 \leq i \leq n$ 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 \( U \) if \( |U| < d_j \). Procedure \( \text{FEAS}(J, b) \) is a slight generalization. It finds a maximum subset of \( J \) that can be scheduled in the interval \([0, b]\). \( \text{DONE}(i) \) is set to -1 if job \( i \) is not selected and is set to a number greater than \( J \) otherwise. If \( \text{DONE}(i) > J \), then job \( i \) is to be scheduled from \( \text{DONE}(i) - 1 \) to \( \text{DONE}(i) \). The procedure itself returns a value that equals the number of jobs selected. The correctness of \( \text{FEAS} \) 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.

```plaintext
line Procedure \text{FEAS}(J, n, b)
// select a maximum number of jobs for processing //
// in \([0, b]\) \n\ begin{tabular}{lll}
 1 set \( J \); integer \( n, b \); global \( \text{DONE}(1:n) \)
 2 sort \( J \) into nondecreasing order of due times
 3 \( \text{DONE}(1:n) \leftarrow -1 \) //initialize//
 4 \( j \leftarrow 0 \)
 5 for \( i \leftarrow 1 \) to \( n \) do
 6 case
 7 : \( j > b \): \text{return}(j) //interval full//
 8 : \( j \leq b \): //select i// \( j \leftarrow j + 1; \text{DONE}(i) \leftarrow j \)
 9 end case
10 end for
11 return(j)
end \text{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 \( D(i) < D(i+1), 1 \leq i < k \). 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 \( D(j) = J \) and \( n(J) = 0 \). Define \( F(i) \) to be the value of \( j \) when procedure \( \text{FEAS} \) (Figure 4.1) has just finished considering all jobs in \( J \) with due time at most \( D_i \). It is evident that:

\[
F(j) = D(j) = 0
\]

(4.1)

\[
F(i) = \min(F(i-1)+n(i), D(i), b), 1 \leq i \leq k
\]

Expanding the recurrence (4.1), we obtain:

\[
F(1) = \min(D(0)+n(1), D(1), b)
\]

\[
F(2) = \min(F(1)+n(2), D(2), b)
\]

\[
= \min(D(0)+n(1)+n(2), D(1)+n(2), b+n(2), D(2), b)
\]
\[-9\]

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

And, in general

\[
F(m) = \min \{ \min \{ D(i) + \sum_{q=i+1}^{m} n(q), b \} \} \quad 0 \leq m \leq k
\]

The maximum number of jobs in \( J \) that can be scheduled in \([0, b]\), \( b > 3 \), 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 \( d(i) > j, 1 \leq i \leq n \). The computation steps are:

1. **Step 1:** sort \( d(1:n) \) into nondecreasing order.
2. **Step 2:** determine the points \( r(3), \ldots, r(k-1) \) in \( d(3:n) \) where the due times change 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 \( D(i) = d(r(i)) \), \( 1 \leq i \leq k \); \( D(0) = a \).
3. **Step 3:** compute \( F(k) = \min \{ n + \min \{ D(i) - r(i), b \} \} \)

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 \( k = 6; r(0:6) = (3, 3, 7, 8, 9, 13, 15) \) and \( D(0:6) = (0, 2, 3, 5, 8, 9, 11) \). So, \( F(k) = \min \{ D(i) - r(i), b \} \)

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 \( d(i) < d(i+1), 1 \leq i \leq n-1 \). 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 \( r(3), r(1), \ldots, r(k) \). This can be done in \( O(\log n) \) time using \( n \) PEs and the data concentration algorithm of [20]. Another data concentration step moves \( d(r(3)), d(r(1)), \ldots, d(r(k)) \) into \( D(0), D(1), \ldots, D(k) \). Using \( k+1 \) PEs, \( D(i) - r(i), 0 \leq i \leq k \) 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 \( \lfloor k/\log k \rfloor \) PEs are needed for this; but using \( k/2 \) PEs is
Figure 4.2

(a) input set of jobs.

(b) jobs sorted in nondecreasing order of due time

Figure 4.3

We have seen how to determine the maximum number of non tardy 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:

\[
\text{DONE}(\delta) = \delta
\]

(4.3) \hspace{1cm} \text{DONE}(i) = \min\{\text{DONE}(i-1)+1, d_i\}, 1<i<n

Solving (4.3), we obtain:

(4.4) \hspace{1cm} \text{DONE}(i) = \min \{d_j+i-j, l<i<n \}

Done(i), 1<i<n may be computed in \(O(\log n)\) 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 \(O(\log n)\) time and requires \(n^2\) PES, the overall time complexity is \(O(\log n)\) 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, 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, 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 \(z_i\) 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: $\lambda \leftarrow \{1\}$

\[
\text{for } i \leftarrow 2 \text{ to } n \text{ do } \\
\text{if } R \cup \{i\} \text{ is feasible then } R \leftarrow R \cup \{i\} \\
\text{end for}
\]

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 $O(n\log n)$. For the parallel version, we reduce the job sequencing with deadlines problem into $2n$ independent feasibility problems. First, we note that if $R_1$ and $R_2$ are feasible subsets of $J$ and if $R_1$ is one with maximum return, then $|R_2| < |R_1|$.

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

**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 $A$ and $B$, we obtain schedules $SA'$ and $SB'$ that contain no tardy jobs. In addition, jobs in $A \cap B$ 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, $j \not\in A$. 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, $|B| < |A|.$

From the sequential algorithm for the job sequencing problem and Theorem 5.1, we may derive a parallel algorithm. Let $T_1(i) = \{j|z_j > z_i \text{ or } (z_j = z_i \text{ and } j < i]\}$ and $T_2(i) = T_1(i) \cup \{i\}$. Consider a schedule for $T_1(i)$ 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 $T_2(i)$. 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.

The 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 $T_2(i)$, $1 \leq i \leq n$. The number of non-tardy jobs in the optimal schedules for $T_1(i)$ and $T_2(i)$ 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. [ ]
Figure 5.4

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 sorted data into $n$ copies, one to be used for each $T_1(i)$ and $T_2(i)$. 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 $T_1(i)$ and then $T_2(i)$. $T_1(i)$ is obtained by having the $j$th 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 $x(i) = P(x_i)$, $1 \leq i \leq n$. 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 $O(1/n)$. 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 $O(1/\log n)$.

### 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 admissible iff $s_i \geq s_{i-1} + p_{i-1}$, $2 \leq i \leq n$. 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:

- $e_i = \max[0, a_i - s_i]$
- $t_i = \max[0, c_i - c_i]$
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 [23]. He obtained an $O(n^2)$ algorithm for the case when:

1. $a_i < a_j$ implies $b_i < b_j$

and

2. $g(\cdot)$ and $h(\cdot)$ are monotone nondecreasing continuous functions such that $g(0) = h(0) = 0$.

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

The algorithm of [16] first finds an admissible schedule $S$ using procedure ADMIS (Figure 6.1). This procedure assumes that the jobs are ordered by target start times (i.e., $a_i < a_{i+1}$) and within start times by target due times (i.e., $a_i = a_{i+1}$ implies $b_i = b_{i+1}$). The maximum lateness, $\Delta$, in $S$ is next computed. 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.

Procedure ADMIS (a, p, s, n)
//jobs are ordered by target start and due times//
\[ s_i \leftarrow a_i \]
\[ \text{for } i \leftarrow 2 \text{ to } n \text{ do} \]
\[ s_i \leftarrow \max[a_i, s_{i-1} + p_{i-1}] \]
\[ \text{end for} \]
\[ \text{end ADMIS} \]

Figure 6.1

$\Delta$ can be computed in $O(\log n)$ 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 $O(\log n)$ time using $n$ PEs. Finally, the $s_s$ can be updated in $O(1)$ time using $n$ PEs. Also, the initial ordering of the jobs may be carried out in $O(\log n)$ time with $n$ PEs. All that remains, is the computation of the admissible schedule. From Figure 6.1, we see that

\[ s_i = a_i \]
\[ (6.1) \quad s_i^* = \max[a_i, s_{i-1} + p_{i-1}], \quad 2 \leq i \leq n \]
Expanding the recurrence (6.1), we obtain

\[ s_i = \max \{ a_j + \sum_{k<j} p_k \}, 1 \leq i \leq n \]

It should be easy to see that using (6.2) and \( O(n^3) \) PEs, one can compute all the \( s_i \)s in \( O(\log n) \) time. We shall devote the remainder of the section to the development of an \( O(\log n) \) 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 \( 2^k \)-blocks of \( A(0:10) \) are \([0,1],[2,3],[4,5],[6,7],[8,9],[10]\) the \( 2^k \)-blocks are \([0,1,2,3],[4,5,6,7],[8,9,10]\) and \([0,1],[2,3]\) are sibling blocks; so also are \([0,1,2,3],[4,5,6,7]\) 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 \([i,i+1,i+2,\ldots,r]\) 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

\[ S(j) = \sum_{q=i}^{j-1} p(q), \quad j \text{ is an index in this block} \]

\[ T(j) = \sum_{q=i}^{r} p(q), \quad r \text{ is the highest index in the block} \]

\[ Q(j) = \max \{ A(q) + \sum_{t=q}^{j-1} p(t) \}, j \text{ is a block index} \]

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

For a \( 2^k \)-block \([i]\), we have:

\[ S(i) = 0; \quad T(i) = P(i); \quad Q(i) = A(i); \quad U(i) = A(i) \]

Let \( X = [i,i+1,\ldots,u] \) and \( Y = [u+1,\ldots,v] \) be two sibling \( 2^k \)-blocks. Their union \( Z = [i,i+1,\ldots,v] \) 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:

\[ S(j) \quad \text{if } j \in X \]

\[ S'(j) = \begin{cases} S(j) + T(i) & \text{if } j \in X \\ S(j) & \text{if } j \in Y \end{cases} \]
\[ T'(j) = T(j) + T(u+1) \text{ if } j \not\in X \]
\[ T'(j) = T(j) + T(i) \text{ if } j \not\in Y \]
\[ Q'(j) = Q(j) \text{ if } j \not\in X \]
\[ Q'(j) = \max(Q(j), U(i)+S(j)) \text{ if } j \not\in Y \]
\[ U'(j) = Q'(v)+P(v) \]

One also notes that with respect to the entire \( \log_2 n \) -block \([0,1,2,\ldots,n-1]\),
\[ Q(j) = \max_{0 \leq q \leq j} \{ a + \sum_{t=q}^{j} p(t) \} \]
\[ = s_j \text{ of (6.2)} \]

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

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
3 & 2 & 2 & 4 & 1 & 3 & 4 & 1 & 3 & 4 \\
A & 0 & 1 & 4 & 8 & 9 & 9 & 15 & 15 & 16 & 17 \\
\end{array}
\]

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 \( 0,1,\ldots,\lceil n/2 \rceil -1 \). With respect to our example of Figure 6.3, when \( k=0 \), PE(0) will compute the new values of \( S(1), T(3), T(1), Q(1), U(3), \) 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 admissible schedule

are both assigned to the new $2^2$-block $[4,5,6,7]$, 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 \( f(i,j) = \ldots i_{j+1}, i_j, i_{j-1}, \ldots i_0 \). When \( 2^k \)-blocks are being combined, PE(\( i \)) computes \( S(f(i,k)+2^k), T(f(i,k)+2^k), Q(f(i,k)+2^k), U(f(i,k)), \) 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, \( S(f(i,k)+2^k) \) of (6.5a) has been replaced by \( S(j)+T(j-2^k) \). Similarly, \( T(j)+T(u+1) \) has been replaced by \( T(j)+T(u+2) \); and \( U(i)+S(j-1) \) by \( U(j)+S(j) \). Note that as a result of this change, new \( T \) and \( U \) values for the rightmost 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 the rightmost block are never used. One may verify that \( \max[U(j+2^k),U(j)] = Q'(v) + P(v) \) (Eq. 6.5d). When \( k = \lfloor \log n \rfloor -1 \), only \( Q \) need be computed.

```
procedure PADMIS(A,P,s,n)
// Obtain the admissible schedule(s)
declare n, A(0:n-1),P(0:n-1),S(n-1),T(0:n-1)
declare Q(0:n-1),U(0:n-1),j,i
for each PE(i) do in parallel
    i = f(i,0)
    // Initialize 2^k-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 \lfloor \log n \rfloor -1 do
        // Combine 2^k-blocks/
        j = f(i,k) // PE assignment/
        if j+2^k < n then
            U(j+2^k) = \max[S(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 \leq i \leq n
end PADMIS
```

Figure 6.4: Parallel admissible schedule algorithm.

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 $s_s$ in $O(\log n)$ 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 [5]. With this grouping technique, the parallel admissible 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 $O(\log n)$ 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 laid 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.](image)

In the mathematical formulation of this problem, we are given $n$ pairs of points $(a_i, b_i), 1 \leq i \leq n$. 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 $(6,3)$; the channel assignment is: $(1,4)$ and $(6,3)$ in channel 1, $(2,5)$ in channel 2, and $(3,7)$ in channel 3.

The job sequencing problem with release times and due times 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
t heir 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 re-
striction on \( r_i, p_i, \) and \( d_i \) is removed, the problem is NP-
nard.

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

\textbf{step 1:} Sort the multiset \( \{a_i \mid 1 \leq i \leq n \} \cup \{b_i \mid 1 \leq i \leq n \} \)
of the 2n end points into nondecreasing order.

\textbf{step 2:} \( m \leftarrow 0; \) stack \( \leftarrow \) empty

\textbf{step 3:} process the 2n points one by one

\hspace{1cm} if the point being processed is an \( a_i \)

\hspace{2.5cm} then if stack empty then \( m \leftarrow m + 1 \)

\hspace{2.5cm} assign this wire run to channel \( m \)

\hspace{2.5cm} else unstack a channel from

\hspace{3.5cm} the stack and assign the wire to this

\hspace{3.5cm} channel

\hspace{1cm} endif

\hspace{1cm} else

\hspace{2.5cm} put the channel used by this wire onto the

\hspace{3.5cm} stack

\hspace{1cm} 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 deter-
mined. It is possible to determine this number without
actually obtaining a channel assignment. Let \( c_1, c_2, \ldots, c_n \)
be the sorted sequence of 2n end points. Set \( z_1 = 1 \) if \( c_i \)
is \( a_1 \) and \( z_i = -1 \) if \( c_i \) is a \( b_i \). It is easy to see that
\( r_j = \sum z_i \) gives the number of wires that either start at \( c_i \)
or cross the point \( c_i \). Further, \( \max \{r_j\} \) 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 algo-
rithm of [6]. 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 as
and bs 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 com-
plexity of \( O(\log n) \) and an EPU of \( O(1/n) \). If the \( O(\log^2 n) \), \( n \)
PE sorting algorithm of [21] is used instead, the time com-
plexity is \( O(\log^2 n) \) and the EPU is \( O(1/\log n) \).
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](image)

Figure 7.2

<table>
<thead>
<tr>
<th>C_1</th>
<th>C_2</th>
<th>C_3</th>
<th>C_4</th>
<th>C_5</th>
<th>C_6</th>
<th>C_7</th>
<th>C_8</th>
<th>C_9</th>
<th>C_{10}</th>
<th>C_{11}</th>
<th>C_{12}</th>
<th>C_{13}</th>
<th>C_{14}</th>
<th>C_{15}</th>
<th>C_{16}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_1</td>
<td>a_2</td>
<td>a_3</td>
<td>b_2</td>
<td>a_4</td>
<td>a_5</td>
<td>b_3</td>
<td>b_4</td>
<td>a_6</td>
<td>b_1</td>
<td>a_7</td>
<td>a_8</td>
<td>b_5</td>
<td>a_9</td>
<td>b_7</td>
<td>b_8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Assigned Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>Partial Sum</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MAX</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3

The actual channel assignment can be obtained from the r_j's (recall that r_j = \frac{1}{2} z_j, 1 \leq j \leq 2n. Assume that c_j corresponds to a_j. Let q be the largest index such that q < j, r_q = r_q - 1, and c_q corresponds to an a (say a_{\alpha_j}). If no such q exists, set q to j. An examination of the algorithm of Gupta et al. reveals that if q = \emptyset, then the channel used by (a_{\alpha_j}, b_j) has not been used earlier. If q \neq \emptyset, then it was most recently used in the interval (a_q, b_q). To see the truth of this, note that at point b_q, the channel assigned to (a_{\alpha_j}, b_j) 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 \( b \). (If \( a_i = a_k \) and \( i < j \), then we say that \( a_i \) is before \( a_k \)). 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 partitioning 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 partitions 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 \( O(\log n) \) time using \( n/\log n \) PEs.

![Diagram showing partitioning](image)

**Figure 7.4: Partitions for Example 7.3**

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

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:

```matlab
for j = 1 to \( \log n \) do
  for each i for which \( Q(i) = 0 \) do in parallel
    if \( L(L(i)) = j \) then \( Q(i) = Q(L(i)) \)
    \( L(i) = L(L(i)) \)
  end for
end for
```

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 \( O(\log n) \) \( n \) PE sorting algorithm); its EPU is \( O(1/n) \).
d. 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

17. McNaughton, R., "Scheduling with deadlines and loss..."


**Title:** Parallel Scheduling Algorithms

**Authors:** Eliezer Dekel and Sartaj Sahni

**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.