Nearly On Line Scheduling Of Multiprocessor Systems With Memories*

Ten-Hwang Lai and Sartaj Sahni
University of Minnesota

Abstract

We show that no multiprocessor system that contains at least one processor with memory size smaller than at least two other processors can be scheduled nearly on line to minimize the finish time. An efficient nearly on line algorithm to minimize $C_{\max}$ is developed for multiprocessor systems that do not satisfy the preceding requirement. Finally, we review the complexity of some other scheduling problems for multiprocessor systems with memories.

Keywords and Phrases

Multiprocessor systems, memories, scheduling, nearly on line, $C_{\max}$, complexity, algorithm.

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

1. Introduction

A *uniform processor system with memories* consists of a set of m, $m \ge 1$, processors P_i, $1 \le i \le m$. A tuple (s_i, μ_i) is associated with each processor P_i. s_i is the *speed* of P_i and μ_i is its memory size. When $\mu_1 = \mu_2 = ... = \mu_m$, the processor system is referred to as a *uniform processor system*. A uniform processor system with memories in which $s_1 = s_2 = ... = s_m$ is also called an *identical processor system with memories*. If both $s_1 = ... = s_m$ and $\mu_1 = \mu_2 = ... = \mu_m$ the processor system is simply a *system of identical processors*.

Let J = $\{J_1, J_2, ..., J_n\}$, be a set of n, $n \ge 1$, independent jobs. With every job, J_i, a 4-tuple (t_i, m_i, r_i, d_i) is associated. t_i , $t_i \ge 0$, is the *processing requirement* of job J_i. So, a processor with a speed of s_j would take t_i/s_j time to completely run job J_i. m_i is J_i's *memory requirement*. J_i can be run (or processed) only on those processors that have a memory size no smaller than m_i. r_i is J_i's *release time*. The processing of J_i cannot begin until time r_i. Finally, d_i is J_i's *due time*. This represents the time by which J_i's processing should complete. If it does not, then J_i is *tardy*.

A *feasible preemptive schedule*, S, for job set J on the processor system P = $\{P_1, P_2, ..., P_m\}$ is an assignment of jobs to time slots on the processors such that:

(a) No job is processed before its release time.

(b) No job is processed by more than one processor at any given time.

(c) No job is assigned to a processor with memory size less than the job's memory requirement.

(d) No processor processes more than one job at any time.

(e) The total processing assignment for each job equals its processing requirement.

If in addition to the above requirements, S is such that each job is processed continuously from its start to its finish on the same processor, then S is a *nonpreemptive* schedule. The finish time, f_i, of J_i is the time at which the processing of J_i is completed. Note that f_i is defined relative to a schedule S. The *length* or *finish time*, $C_{\max}(S)$, of schedule S is the least time by which all jobs have been processed. So, $C_{\max}(S) = \max_i\{f_i\}$. The $C_{\max}$ problem is that of finding a schedule S that minimizes $C_{\max}$.

The lateness, L_i, of job J_i in schedule S is $f_i - d_i$. The maximum lateness, $L_{\max}$, of any job in S is $\max_i\{L_i\}$. The $L_{\max}$ problem is that of finding a schedule S with minimum $L_{\max}$.

Any algorithm that produces feasible preemptive schedules is called a *scheduling algorithm*. A scheduling algorithm that generates schedules with minimum $C_{\max}$ is an *optimal* scheduling algorithm. A scheduling algorithm that generates the schedule from time 0 to time t (for every t) only using information about jobs released before t is called an *on line algorithm*. If in addition to knowning the jobs released before t, the algorithm also needs to know the next release time (either t or following t), then the algorithm is *nearly on line*. A scheduling algorithm that is not nearly on line is *off line*.

The $C_{\max}$ problem is known to be NP-hard when S is required to be a nonpreemptive schedule. This is true even for processor systems with m = 2, $s_1 = s_2$, $\mu_1 = \mu_2$ and job sets with $r_1 = r_2 = ... = r_n$ [5]. Hence, we shall be concerned primarily with schedules in which preemptions are permitted.

For identical processor systems (i.e., $s_1 = s_2 = ... = s_m$ and $\mu_1 = \mu_2 = ... = \mu_m$) there is an off line algorithm with complexity O(nlogmn) that obtains schedules (if they exist) with a given $C_{\max}$ [11]. Bruno and Gonzalez [4] have developed an O(mn + nlogn) nearly on line algorithm that obtains optimal schedules for identical processor systems.

If P is a uniform processor system (i.e., $\mu_1 = \mu_2 = ... = \mu_m$) and all jobs have the same release time, minimum $C_{\max}$ schedules can be obtained in O(n + mlogm) time using the algorithm of Gonzalez and Sahni [6]. When the release time are not necessarily the same, the off line algorithm of Sahni and Cho [12] may be used to obtain schedules (if they exist) with a given $C_{\max}$ in O(mn + nlogn) time. If a nearly on line algorithm is desired, the algorithm of [13] is nearly on line and generates optimal schedules in O(m^2n + mnlogn) time. The algorithms of [12] and [13] generate schedules with O(mn) preemptions. There is another nearly on line algorithm for uniform processor systems. This is due to Labetoulle et al. [8] and generates optimal schedules with O(n^2) preemptions in O(n^2) time.

The problem of scheduling systems of identical processors with memories has been studied by Kafura and shen [7] and Lai and Sahni [9]. Kafura and Shen [7] develop an O(nlogm) ($n \geq m$) algorithm for the $C_{\max}$ problem when all jobs have the same release time. Lai and Sahni [9] consider the $L_{\max}$ problem when all jobs have the same release time. The algorithm they develop has complexity O(kn^2 + nlogn) where k is the number of distinct due times. Their algorithm can also be used to solve the $C_{\max}$ problem when the release times are not necessarily the same. When used for this problem, their algorithm is off line and has the same complexity as for the $L_{\max}$ problem except that now k is the number of distinct release times.

The general problem of scheduling systems of uniform processors with memories has been studied by Lai and Sahni [10]. They obtain linear programming formulations for both the $C_{\max}$ and $L_{\max}$ problems. The proposed algorithms are off line. In addition, they also develop low order polynomial time algorithms for special classes of processor systems when all jobs have the same release time.

In this paper, we examine the problem of obtaining nearly on line algorithms for uniform processor systems with memories. We may partition the set of processors in a processor system into two partitions A and B such that B contains all processors with the least memory and A contains the remaining processors. I.e., if the memory sizes μ_1, μ_2, μ_3, μ_4, and μ_5 of a five processor system are 10, 20, 10, 15, and 10, respectively, then B = $\{P_1, P_3, P_5\}$ and A = $\{P_2, P_4\}$. In section 2, we show that whenever $|A| \geq 2$, no nearly on line algorithm exists. In section 3, we develop a fast nearly on line algorithm for the case |A| = 1. When |A| = 0, the processor system is simply a uniform processor system and a nearly on line algorithm for such systems already exists [8, 13]. Finally, in section 5 we review the complexity of some scheduling problems for systems of processors with memories.

2. $|A| \geq 2$

In this section, we show that whenever a uniform processor system contains at least two processors that have a memory size larger than the smallest memory size in the system, no nearly on line algorithm is possible.

To get a flavor for the proof, we first establish this result for the processor system $\{P_1, P_2, P_3\}$ with $s_1 = s_2 = s_3$, and $\mu_1 = \mu_2 > \mu_3$. Suppose that four jobs are released at time 0. Their processing times are 1, 1, 3, and 3 respectively. The memory requirements are μ_1, μ_1, μ_3, and μ_3 respectively. The next release time is 1. The schedule from 0 to 1 must be constructed without any knowledge of the jobs to be released at or after time 1. Let us consider two possible schedules for this time interval. In the first of these, jobs 1 and 2 are the only jobs scheduled on P_1 and P_2 from 0 to 1. Jobs 3 and 4 are used to utilize P_3. The resulting schedule is as in Figure 1(a). In the second schedule (Figure 1(b)), jobs 1 and 2 are assigned to P_1 to utilize all of P_1's capacity from 0 to 1. Jobs 3 and 4 are assigned equally to P_2 and P_3.

Figure 1

Note that if only 1 job with processing requirement 1 and memory requirement μ_1 is released at time 1, then a schedule of length 3 can be obtained from Figure 1(b). This is optimal. From the schedule of Figure 1(a), we can at best obtain a schedule of length 3.5. Furthermore, all schedules of length 3 have the form of Figure 1(b) from 0 to 1. On the other hand, if two jobs, each with a processing requirement of 5 and memory requirement of μ_1, are released at time 1, then all schedules having a length of 6 have the form of Figure 1(a) from 0 to 1. This is the optimal length. The best schedule that can be obtained from Figure 1(b) has a length of 6.5.

Since, the nature of the jobs to be released at time 1 is not known in advance, there is no way to determine the form of the schedule from 0 to 1 in order to guarantee a schedule with minimum $C_{\max}$. Hence, there is no nearly on line algorithm for the processor system constructed above.

Theorem 1: Let $\{P_1, P_2, ..., P_m\}$, m ≥ 2, be an arbitrary system of m processors with memory sizes $\mu_1, \mu_2, ..., \mu_m$, and speeds $s_1, s_2, ..., s_m$, respectively. Let $\mu = \min_i\{\mu_i\}$; A = $\{j \mid \mu_j > \mu\}$; and B = $\{j \mid \mu_j = \mu\}$. If $|A| \geq 2$, then there is no nearly on line algorithm that minimizes $C_{\max}$ for this processor system.

Proof: Without loss of generality, we may assume that 1, 2 ε A and 3 ε B; $s_2 \leq s_1 \leq s_j$ for all j, j ε A - {1, 2}; and that $s_3 \leq s_j$ for all j, j ε B. So, P_1 and P_2 are the two slowest processors with memory larger than μ and P_3 is the slowest processor with memory equal to μ.

Let $\Delta_1 = s_2/s_1$; $\Delta_2 = (s_2/s_1)^2$; and $f = 1 + (1 + \Delta_1 + \Delta_2)(1 + s_2/s_3)$. Define the m - 3 jobs J_i, $4 \leq i \leq m$ such that $t_i = s_i f$ and $m_i = \mu_i$. Assume that these jobs are released at time 0. Let R denote the set of remaining jobs released at or after time 0. Assume that the optimal schedule for $R \cup \{J_i \mid 4 \leq i \leq m\}$ has $C_{\max} = f$. It should be clear that if R contains no job with memory requirement larger than $\min\{\mu_1, \mu_2\}$, then there is no advantage to having a schedule in which jobs from R are scheduled on $P_4, P_5, ..., P_m$. So, we may assume that the jobs J_i, $4 \leq i \leq m$, in J are scheduled to fully utilize $P_4, P_5, ..., P_m$ and that P_1, P_2, P_3 are fully available for R. Hence, we need only concern ourselves with job set R and processors P_1, P_2, and P_3.

Now suppose that R contains 6 jobs with $t_1 = s_1$, $t_2 = s_2$, $t_3 = (1 + \Delta_1 + \Delta_2)s_3$, $t_4 = (1 + \Delta_1 + \Delta_2)s_2 + s_3$, $t_5 = [(t_3 + t_4)/s_3 - 1]s_1$, and $t_6 = [(t_3 + t_4)/s_3 - 1]s_2$. Assume that $m_1 = m_2 = m_5 = m_6 = \min\{\mu_1, \mu_2\}$; $m_3 = m_4 = \mu$; jobs 1, 2, 3, and 4 are released at 0; and jobs 5 and 6 are released at 1. All schedules with finish time f have the form given in Figure 2(a).

Next, suppose that R contains 8 jobs with the first four being as before. Let $\Delta_3 = (1 + \Delta_1 + \Delta_2)s_2/s_3$ and let $t_5 = \Delta_2 s_1$, $t_6 = (1 + \Delta_3)s_1$, $t_7 = (1 + \Delta_3)s_2$, and $t_8 = \Delta_3 s_3$. Assume that $m_5 = \min\{\mu_1, \mu_2\}$; $m_6 = m_7 = m_8 = \mu$; job 5 is released at time 1; and jobs 6, 7, and 8 are released at time 1 + $\Delta_1 + \Delta_2$. A minimum $C_{\max}$ schedule for this set of 8 jobs is given in Figure 2(b). This schedule

Figure 2

has length f. It is easily verified that there is no schedule for this set of 8 jobs that both has a finish time of f and in which jobs 1 and 2 are scheduled as in Figure 2(a). To see this, observe that $t_5/s_1 = \Delta_2 < \Delta_1 + \Delta_2$ and $t_5/s_2 = \Delta_2 s_1/s_2 = \Delta_1 < \Delta_1 + \Delta_2$. Hence, no matter how job 5 is scheduled on P_1 and P_2 from 1 to $1 + \Delta_1 + \Delta_2$ (Figure 2(c)), there must be intervals in which both

P_1 and P_2 are idle. Simultaneous idle times on P_1, P_2, and P_3 cannot be filled up by jobs 3 and 4 alone. Hence, no matter how jobs 3, 4, and 5 are scheduled on P_1, P_2, and P_3 from 1 to $1 + \Delta_1 + \Delta_2$ (in Figure 2(c)), there must be some idle time. Consequently the overall schedule length must exceed f.

Since at time 0, there is no way to distinguish between the two job sets of the previous paragraphs, there is no nearly on line algorithm for the given processor system that minimizes $C_{\max}$. □

3. |A| < 2

When |A| = 0, all processors have the same memory size and the nearly on line algorithm of Sahni and Cho [13] may be used to minimize $C_{\max}$. So, we need only consider the case |A| = 1. In this case, the m processor system consists of m - 1 processors having the same memory size μ and 1 processor with memory size larger than μ.

Let $\{P_1, P_2, ..., P_m\}$ be an m processor system with |A| = 1. Assume that the processors have been indexed such that $s_1 \geq s_2 \geq ... \geq s_m$ and that P_k is the lone processor with memory size larger than μ. We may arrive at a nearly on line scheduling algorithm, by first determining how jobs with memory requirement larger than μ are to be scheduled.

Suppose that n jobs are to be scheduled and that their release times are $r_1, r_2, ..., r_n$, respectively. Let $c_1, c_2, ..., c_u$ be the distinct release times in the multiset $\{r_1, r_2, ..., r_n\}$. We assume that $c_1 < c_2 < c_3 < ... < c_u$. Let R_i denote the set of jobs with release time c_i and having memory requirement larger than μ. Let S_i be the set of jobs with release time c_i and memory requirement μ. Let T_i be the sum of the processing requirements of the jobs in R_i. We shall show that there is always a minimum $C_{\max}$ schedule in which the jobs in R_i are scheduled from τ_i to $\delta_i = \tau_i + T_i/s_k$, where τ_i is as given below:

$$\tau_i = \begin{cases} c_1 & i = 1 \\ \max\{c_i, \delta_{i-1}\} & i > 1 \end{cases} \qquad (1)$$

Consider any minimum $C_{\max}$ schedule for the given n jobs. Clearly, all jobs in $\bigcup_{i=1}^{u} R_i$ must be scheduled on P_k. Suppose that the jobs in $\bigcup_{i=1}^{u} R_i$ are not scheduled as discussed above. Let r be such that there are no preemptions in the interval (jr, (j+1)r) for any j. Further, no c_i, τ_i, or δ_i is in the interval (jr, (j+1)r) for any j. (Note that r does exist since all values we deal with are rational numbers.) The time interval 0 to $C_{\max}$ may be divided into intervals of length r. These intervals will be called r-intervals. Let a be the least i such that the interval $[\tau_i, \delta_i]$ has a job not in R_i scheduled on P_k. Let b be the leftmost r-interval in $[\tau_a, \delta_a]$ such that the job scheduled on P_k in this interval is not in R_a. Let c be the leftmost r-interval to the right of $[\tau_a, \delta_a]$ such that the job

Figure 3

scheduled on P_k in this interval is from R_a. (Note that no job in R_a can be scheduled to the left of $[\tau_a, \ \delta_a]$.) Figure 3 shows two possible situations for b and c. In the first (Figure 3(a)), both r-intervals lie between two consecutive release times. In this case, we merely interchange the scheduling assignments of the two r-intervals. The resulting schedule satisfies the release time requirements. The second possibility is that at least one release time falls between the two r-intervals (Figure 3(b)). In this case, a straightforward interchange of the two r-intervals could result in some jobs being scheduled before their release times. Assume that at least one of the jobs scheduled in the r-interval c has a release time greater than c_i. In this case, the interchange proceeds as follows. First interchange the jobs scheduled in the r-intervals b and c on P_k (jobs 1 and 2 of Figure 3(b)). If job 1 was not previously scheduled in c, no conflict is created and we are done. If job 1 was already scheduled in c, a conflict is created. The earlier scheduling of 1 in c is exchanged with the job scheduled on the same processor in b, i.e., job 3 of Figure 3(b). If job

3 was not already scheduled in c, we are done with the interchange. If it was, then the earlier scheduling of 3 in c is exchanged with job 4 scheduled on the same processor in b. And so on. This exchanging process is clearly finite and has the result of producing a new schedule that does not violate any of the release time requirements.

By continuing in the way described above, the original schedule may be transformed into another schedule that has the same $C_{\max}$ and in which all jobs in R_i are scheduled on P_k from τ_i to $\tau_i + T_i/s_k$, $1 \le i \le u$. Scheduling jobs in R_i in this way is easily done on line.

Figure 4

Our nearly on line scheduling algorithm will construct the schedule in u phases. In phase i, the schedule from c_i to c_{i+1}, $1 \le i < u$, will be constructed. In phase u, the minimum $C_{\max}$ is first computed. Let this value be c_{u+1}. Next, all remaining jobs are scheduled in the interval $[c_u, c_{u+1}]$. The scheduling in phase i is done by first computing τ_i using equation (1). Next, all jobs released at c_i and having memory requirement larger than μ are scheduled from τ_i to $\delta_i = \tau_i + T_i/s_k$ on processor k. If $\delta_i \ge c_{i+1}$, then P_k is not available for additional work in the interval $[c_i, c_{i+1}]$ and the schedule for the remaining processors is obtained using one phase of the nearly on line algorithm of Sahni and Cho [13]. If $\delta_i < c_{i+1}$, then P_k is available for further processing from δ_i to c_{i+1} (Figure 4(a)).

Let G_i, $1 \le i \le m$, be m processors with the same memory size. Let $\sigma_i(t)$ be the speed of G_i at time t. $\{G_1, G_2, ..., G_m\}$ is a *generalized processor system* (GPS) [12] iff each $\sigma_i(t)$ is a nondecreasing function of time and $\sigma_i(t) \ge \sigma_{i+1}(t)$, $1 \le i < m$ for all t.

since the remaining jobs to be scheduled in $[c_i, c_{i+1}]$ have the same memory requirement μ, we may ignore the fact that P_k has a larger memory size. Hence, scheduling on the processor system of Figure 4(a) is equivalent to scheduling on the GPS of Figure 4(b). $\sigma_i(t)$ for G_i is defined as below (we assume $s_{m+1} = 0$ for convenience):

$$\sigma_i(t) = \begin{cases} s_i \;, [1 \le i < k] \textit{ or } [k \le i \le m \text{ and } \delta_i \le t \le c_{i+1}] \\ s_{i+1} \;, k \le i \le m \text{ and } c_i \le t \le \delta_i \end{cases} \tag{2}$$

Suppose that at time c_i there are r jobs from $\bigcup_{j=1}^{i} S_j$ that have a nonzero remaining processing requirement. Index these r jobs 1, 2, ..., r and let v_i denote the remaining processing requirement of job i. We assume that the indexing was done such that $v_1 \ge v_2 \ge ... \ge v_r$. We may determine if all these jobs can be completed on the GPS of Figure 4(b) by using the following result from [12].

Theorem 2 [Sahni and Cho]: Let $\{G_1, G_2, ..., G_m\}$ be a GPS and let $\sigma_i(t)$ be the speed of G_i at time t. Let $\{J_1, ..., J_n\}$ be n jobs and let t_i be the processing requirement of job J_i. Assume that $t_1 \ge t_2 \ge ... \ge t_n$ and that n ≥ m (if n < m we may introduce m - n jobs with zero processing requirements). Let $L_i = \sum_{j=1}^{i} t_j$, 1 ≤ i < m and $L_m = \sum_{j=1}^{n} t_j$. The given n jobs can be scheduled on the given GPS to complete by time d iff

$$L_k \le \sum_{j=1}^{k} \int_0^d \sigma_j(t)dt \;,\; 1 \le k \le m \tag{3}$$

If the $v_i s$ and the GPS of Figure 4(b) satisfy (3) with $\int_0^d$ replaced by $\int_{c_i}^{c_{i+1}}$, then all r jobs may be scheduled to complete by c_{i+1}. If (3) is not satisfied, then we need to determine the amount w_j, $w_j \le v_j$, of job j that is to be scheduled in $[c_i, c_{i+1}]$. These w_js can be obtained using the equalizing rule given in [13]. This rule computes the w_is in such a way that

(a) $w_j \le v_{j,}\; 1 \le j \le r$

(b) $w_j \ge w_{j+1},\; 1 \le j < r$

(c) $v_j - w_j \ge v_{j+1} - w_{j+1},\; 1 \le j < r$

(d) $L_q \le \sum_{j=1}^{q} \int_{c_i}^{c_{i+1}} \sigma_j(t)dt,\; 1 \le q \le m$

where $L_q = \sum_{j=1}^{q} w_j$, 1 ≤ q < m (if r < n then set $w_{r+1} = w_{r+2} = ... = w_n = 0$) and $L_m = \sum_{j=1}^{r} w_j$.

(e) $\sum_{j=1}^{q}(v_j - w_j)$ is minimized subjected to the conditions (a) - (d) above for every q, $1 \le q \le \min\{m, r\}$.

Note that algorithm EQUAL of [13] computes the w_is only for a system of uniform processors. It is, however, easily modified to do the same for the GPS of Figure 4(b). Furthermore, lemmas 2.1, 2.2, 2.3, and Theorem 2.1 of [13] hold in the case of a GPS also (though in [13] the proof is provided explicitly only for a uniform processor system). This guarantees the success of our u phase scheduling algorithm.

All that remains is the computation of c_{u+1}. This is done using Theorem 2 with the t_is being the remaining processing times of the jobs at time c_u. Given the simple nature of the GPS of Figure 4(b), the least δ’, $\delta'_{\min}$, such that

$$L_k \le \sum_{j=1}^{k} \int_{c_u}^{\delta_u + \delta'} \sigma_j(t)dt, \ 1 \le k \le m$$

is easily computed. The minimum $C_{\max}$ is $c_{u+1} = \delta_u + \delta'_{\min}$.

The actual scheduling of the w_js in any interval $[c_i, c_{i+1}]$ may be done using the GPS scheduling algorithm of [12]. Once again, since the GPS of Figure 4(b) is quite close to being a uniform processor system (see Figure 4(a)), the scheduling of the w_is may be done in a somewhat simpler manner by extending the algorithm of Gonzalez and Sahni [6] to the processor system of Figure 4(a).

5. Complexity Issues

Published research on the scheduling of multiprocessor systems with memories has been exclusively concerned with the scheduling of independent jobs to minimize either $C_{\max}$ or $L_{\max}$ ([7], [9], and [10]). When precedence constraints may exist amongst the jobs, the $C_{\max}$ problem is NP-hard even when m = 2, $s_1 = s_2$, all jobs require one unit of processing time, and the precedence constraint is as simple as a set of chains. This follows from the knowledge that the $C_{\max}$ problem with m = 2, unit processing times, chain precedence, and one resource of capacity 1 is NP-hard [3]. To see this, observe that when one processor has a memory size larger than the other (in a 2 processor system), memory is equivalent to a single resource of size 1 (the job running on the processor with larger memory is considered to be using the resource while the job running on the other processor is not using the resource).

Another NP-hard result is a direct consequence of Blazewicz’s [1] result that when m = 2, $s_1 = s_2$, $\mu_1 = \mu_2$ and a single resource with capacity 1 is available, the problem of minimizing the mean flow time $((1/n)\sum f_i)$ is NP-hard. From this result, we see that minimizing the mean flow time when m = 2, $\mu_1 > \mu_2$, and $s_1 = s_2$ is NP-hard.

6. Conclusions

We have obtained a sharp boundary between the multiprocessor systems for which nearly on line scheduling algorithms that minimize $C_{\max}$ exist and those for which such algorithms do not exist. A polynomial time nearly on line algorithm to minimize $C_{\max}$ on those systems for which this is possible has also been obtained. Finally, we have pointed out the similarity between multiprocessor systems with memories and those with a single resource of capacity one.

References

1. J. Blazewicz, "Mean flow time scheduling under resource constraints," Preliminary Report 19/77, Institute of Control Engineering, Poznan, Poland, 1977.

2. J. Blazewicz, "Scheduling with deadlines and resource constraints," Preliminary Report PR-25/77, Institute of Control Engineering, Poznan, Poland, 1977.

3. J. Blazewicz, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," Report BW-127/80, Department of Operations Research, Mathematical Center, Amsterdam, 1980.

4. J. Bruno and T. Gonzalez, "A New Algorithm for Preemptive Scheduling of Trees," *JACM*, Vol. 27, No. 2, PP. 287-312, 1981.

5. M.R. Garey and D.S. Johnson, "Computer and intractability, a guide to the theory of NP-completeness," W.H. Freeman and Co., San Francisco, 1979.

6. T. Gonzalez and S. sahni, "Preemptive scheduling of uniform processor sustems," *JACM*, Vol. 25, 1978, PP. 92-101.

7. D.G. Kafura and V.Y. Shen, "Task scheduling on a multiprocessor system with independent memories," *SICOMP*, Vol. 6, No. 1, 1977, PP. 167-187.

8. J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Preemptive scheduling of uniform machines subject to release dates," Report BW99, Department of Operations Research, Mathematical Center, Armsterdam, 1977.

9. T.H. Lai and S. Sahni, "Preemptive scheduling of a multiprocessor system with memories to minimize $L_{\max}$," Report No. 81-20, Computer Science Dept., University of Minnesota, Minneapolis, 1981.

10. T.H. Lai and S. Sahni, "Preemptive scheduling of uniform processors with memory," Report No. 82-5, Computer Science Dept., University of Minnesota, Minneapolis, 1981.

11. S. Sahni, "Preemptive scheduling with due dates," *OP RES*, Vol. 27, No. 5, 1979, PP. 925-934.

12. S. Sahni and Y. Cho, "Scheduling independent tasks on a uniform processor system," *JACM*, Vol. 27, No. 3, 1980, PP. 550-563.

13. S. Sahni and Y. Cho, "Nearly on line scheduling of a uniform processor system with release times," *SICOMP*, Vol. 8, No. 2, 1979, PP. 275-285.