ALGORITHMS FOR TWO SCHEDULING PROBLEMS

Jayme Luiz Szwarcfiter
NCE/UFRJ
Andréa Werneck Richa
NCE/UFRJ
Algorithms for Two Scheduling Problems

Jayme Luiz Szwarcfiter     Andréa Werneck Richa

Julho de 1992

Resumo
Descrevemos algoritmos para resolver os dois problemas de escalonamento envolvendo processadores paralelos idênticos, que se seguem. Cada tarefa necessita de uma unidade de tempo de processamento, tem uma data de chegada e um peso associados. O primeiro problema também envolve a existência de prazos e consiste em minimizar o somatório ponderado das tarefas tardias. Já o segundo problema consiste em se minimizar o somatório ponderado dos tempos de término das tarefas. Os algoritmos propostos rodam em tempos $O((1 + \log m)n^2/m)$ e $O((\log n + n/m)n)$, respectivamente.

Abstract
We describe algorithms for solving the following two scheduling problems on identical parallel processors. Each job requires unit processing time, has a release date and a weight. The first problem also involves the existence of deadlines and consists of minimizing the weighted sum of tardy jobs. The second consists of minimizing the weighted sum of completion times. The proposed algorithms run in time $O((1 + \log m)n^2/m)$ and $O((\log n + n/m)n)$, respectively.
1 Introduction

In this paper we are concerned with two particular scheduling minimization problems involving unit processing time jobs with release dates and weights and identical parallel processors. The former problem also involves the existence of deadlines and consists of minimizing the sum of weighted tardy jobs. The latter consists of minimizing the sum of weighted completion times. We present algorithms with time complexities $O((1 + \log m)n^2/m)$ and $O(n(\log n + n/m))$ respectively, for finding the corresponding minimum schedulings.

Variations and related problems to those here considered can be found in Blazewicz [1, 2, 3], Frederickson [5], Garey et al [6], Graham et al [7], Kawaguchi and Kyan [8] and Lawler [9, 10]. It should be mentioned that the two problems here considered can be solved as special cases of the general method given in [9] in $O(n^3)$ time.

We assume that all release dates and deadlines are integer numbers. If only the release dates (or deadlines) are integers, then we can easily modify the deadlines (or release dates) to suitable integers, transforming the original problem into an equivalent one. If neither the release dates nor the deadlines are composed solely by integers, then the problems are considerably more difficult. Indeed the case in which a polynomial time algorithm exists so far, is restricted to a fixed number of processors (see Carlier [4] or Garey et al [6]).

2 Definitions and Notation

Let $J = \{J_1, \ldots, J_n\}$ be a set of $n$ independent jobs, each job having a release date $r_i$, a deadline $d_i$, a weight $w_i$, and unit processing time. Let $P = \{P_1, \ldots, P_m\}$ be a set of identical parallel processors, $1 \leq m \leq n$. A scheduling $S$ for $(J, P)$ is an injective function $S : J \rightarrow (\mathbb{Z}, P)$, which assigns to each job $J_i \in J$ an integer $s_i \in \mathbb{Z}$—called the starting time of $J_i$—and a specified processor $P_j \in P$. We say that $J_i$ has been scheduled at time $s_i$ and processor $P_j$. The value $C_i(S) = s_i + 1$ (or simply $C_i$) is called the completion time of $J_i$ in $S$. A job not yet assigned the pair $(s_i, P_j)$ is called unscheduled; otherwise it is called scheduled. A time $t$ is called available if there exist at least one processor $P_j$ such that no job is scheduled at time $t$ and processor $P_j$.

A schedule $S$ that satisfies $r_i \leq s_i$ and $C_i \leq d_i$, for every job $J_i \in S$ is said to be feasible. Given a feasible schedule and a job $J_i$ possibly unscheduled, define a chain from $J_1$ to $J_k$ as a sequence $J_1, \ldots, J_k$ such that $k > 1$ and

(i) $J_2, \ldots, J_k$ are all scheduled jobs, and

(ii) $r_i \leq s_{i+1} < d_i$, $1 \leq i < k$.

We also say that $J_1$ reaches $J_k$ at time $s_k$. The value $|s_k - s_2|$ is called the length of the chain. A back chain is a chain in which additionally $s_i > s_{i+1}$, for $1 < i < k$. If $J_1$ is a scheduled job, these inequalities ought to hold also for $i = 1$. 

Let $J_i$ and $J_k$ be two scheduled jobs in $S$ such that $s_i > s_k$. It follows that if there exists a chain from $J_i$ to $J_k$ then there exists at least one back chain from $J_i$ to $J_k$. Intuitively, a chain is a sequence of jobs such that every job may be replaced in the scheduling by its successor in the sequence, still maintaining the feasibility of the scheduling. This suggests the operation below.

Given a chain $J_1 \ldots J_k$ from $J_1$ to $J_k$ in the scheduling $S$, we define a *chain replacement*, denoted by $J_1 \leadsto J_k$, as follows:

$$J_i \text{ replaces } J_{i+1} \text{ in } S, \quad 1 \leq i < k$$

i. e., $s_i = s_{i+1}, \quad k > i \geq 1$. This operation preserves the feasibility of $S$ and leaves open the status of $J_k$. See Figure 1.

If $s_i < d_i$ then $J_i$ is called *on time*, otherwise $J_i$ is called *tardy* (see Figure 1). We define the value $u_i = 0$ if $J_i$ is on time, and $u_i = 1$ if $J_i$ is tardy.

Call a schedule minimum if it minimizes a desired optimization criteria.

### 3 Minimizing the Sum of Weighted Tardy Jobs

In this section, we present an $O((1 + \log m)n^2/m)$ time algorithm for minimizing $\sum w_iu_i$ which also minimizes $\sum C_i$ (see Theorem 2).

```plaintext
algorithm $\sum w_iu_i$
data $J = \{J_1, \ldots, J_n\}$, a set of unit processing time jobs, each $J_i$ with release date $r_i$, deadline $d_i$ and weight $w_i$; $P$, $|P| = m$, a set of identical processors;
Order $J$ in non decreasing values of deadlines. Let $J_1, \ldots, J_n$ be such an ordering;
Let $JT$ be the set of tardy jobs;
$JT := \emptyset$
for $i = 1, \ldots, n$ do
  if there exists an available time $t$ such that $r_i \leq t < d_i$ then
    $s_i := t$
  else
    Chain($J_i$)
end for
for $J_i \in JT$ do
  Find smallest available time $t$ such that $r_i \leq t$;
  $s_i := t$
end for

procedure Chain($J_i$)
  Find chain $J_i \ldots J_k$ such that $J_k$ is the (on time) job with least weight reachable from $J_i$;
  if $w_i \leq w_k$ then
    Insert $J_i$ in $JT$
  else
    Perform a (back) chain replacement $J_i \leadsto J_k$ and insert $J_k$ in $JT$;
end procedure
```

2
Theorem 1  The schedule found by the above algorithm minimizes $\sum w_iu_i$.

Proof: Let $SO_i$ and $JT_i$ be respectively the schedule of the on time and the set of tardy jobs found by algorithm $\sum w_iu_i$, after completion of the $i$-th iteration of its first for loop. Suppose by induction that the sum of weights of $JT_{i-1}$ is minimum and consider job $J_i$.

If we find an available time $t$ such that $r_i \leq t < d_i$, then $JT_i = JT_{i-1}$ and the theorem is proved. Else we must consider whether (a) if it is worth scheduling $J_i$ on time instead of another on time job $J_k$, $i \neq k$, or (b) it is not worth doing so.

We will be able to schedule $J_i$ on time if and only if we have at least one back chain from $J_i$ to some $J_r$ ($J_r$ an on time job). Among all those chains, the one that would lead to a minimum $JT_i$ when a chain replacement is performed (if case (a) applies) is the one that turns $J_k$ into a tardy job, where $w_k = \min\{w_r | J_i$ reaches $J_r \in SO_{i-1}\}$. Then $JT_i = JT_{i-1} \cup \{J_k\}$. If case (b) applies, then a minimum $JT_i$ equals $JT_{i-1} \cup \{J_i\}$.

A minimum $JT_i$ will minimize $\{\sum_{j \in JT_i} w_ju_j, JT_i \in \{JT_{i-1} \cup \{J_k\}, JT_{i-1} \cup \{J_i\}\}\}$. Thus, it will be worth scheduling $J_i$ on time if and only if $u_i > w_k$ and the theorem is proved.

Theorem 2  The algorithm $\sum w_iu_i$ also minimizes $\sum C_i$.

Proof: An intuitive algorithm for solving the minimization problem $\sum C_i$ for $(J, P)$ is to schedule the jobs in any given ordering at the smallest available time $t$ such that $r_i \leq t$. This is equivalent to algorithm $\sum w_iu_i$. As the chain replacement eventually performed by procedure $Chain(J_i)$ would be equivalent, in the view of minimizing $\sum C_i$, to consider the jobs $\{J_i, J_j, \ldots, J_k\}$, where $J_iJ_j\ldots J_k$ is the chain found by $Chain(J_i)$, in the relative ordering $J_j\ldots J_iJ_k$ instead of $J_k\ldots J_jJ_i$.

Lemma 1  Let $p$ be the length of the longest possible back chain $C$ from job $J_i$ found by $Chain(J_i)$. Then $p = O(n/m)$.

Proof: Let $SO_i$ be the scheduling of the on time jobs constructed by algorithm $\sum w_iu_i$, at the moment $Chain(J_i)$ is initiated. Let $C = J_iJ_j\ldots J_k$. Then there is no available time $t$ such that $s_k \leq t < s_j$. Otherwise, let $J_q$ be the job in $C$ such that $t = s_q$ is an available time, and $J_p$ the job immediately preceeding $J_q$ in $C$. Thus, $r_p \leq t < d_p$ and $s_p > t$ ($C$ is a back chain). This leads to a contradiction as, by algorithm $\sum w_iu_i$, we would have scheduled $J_p$ at time $t$. Since there are $O(n)$ jobs in the interval $[s_k, s_j]$ and $m$ jobs in parallel, we conclude that $s_j - s_k = O(n/m)$. Thus $p = s_j - s_k + 1$ and the lemma is proved.

For the implementation of procedure $Chain(J_i)$, we could use two heaps for each used time $t$ (that is $t = s_i$ for some $J_i$). The first would inform which job scheduled at $t$ has the smallest release time. The second would inform which one has the smallest release date. Therefore it is possible to find a back chain from $J_i$ to the job $J_k$ with least weight.
in time proportional to the length of the longest possible back chain, i.e., \( O(n/m) \). The chain replacement, clearly, can be made in time proportional to the length of the chain. Updating each pair of heaps requires \( O(\log m) \) time, since the size of each heap is at most \( m \). Therefore \( O(n/m)O(\log m) \) time is required for the processing of each call \( \text{Chain}(J_i) \). There are \( O(n) \) calls of the procedure. The remaining steps require \( O(n \log n) \) time. When \( m = 1 \), the algorithm runs in \( O(n^2) \) time. Therefore the overall bound is \( O((1 + \log m)n^2/m) \).

4 Minimizing the Sum of Weighted Completion Times

We will now present an algorithm for minimizing \( \sum w_iC_i \), which runs in \( O(n(\log n + n/m)) \) time. (It follows trivially that the algorithm also minimizes \( \sum C_i \).)

**Algorithm:** \( \sum w_iC_i \)

**Data:** \( J = \{J_1, \ldots, J_n\} \), a set of unit processing time jobs, each \( J_i \) with release date \( r_i \) and weight \( w_i \); \( P, |P| = m \), a set of identical processors;

Order \( J \) according to non-decreasing values of release dates. Let \( J_1, \ldots, J_n \) be such an ordering;

Let \( S_j = \{J_i | r_i = t_j\} \), \( 1 \leq j \leq k \), where \( t_1 < \ldots < t_k \) are the distinct values of \( r_i \), \( 1 \leq i \leq n \);

for \( j := 1, \ldots, k \) do

Order \( S_j \) in non-decreasing order of \( w_i, J_i \in S_j \). Let \( J_{j1}, \ldots, J_{jS_j} \) always denote
the elements of \( S_j \) in such an ordering;

\( j := 1 \)

while \( j \leq k \) do

if \( |S_j| \leq m \) then

\( s_{ji} := t_j, 1 \leq i \leq |S_j| \)

\( j := j + 1 \)

else

\( s_{ji} := t_j, 1 \leq i \leq m \)

\( S_j := S_j - \{J_{j1}, \ldots, J_{jm}\} \)

if \( t_{j+1} = t_j + 1 \) then

\( S_{j+1} := \text{Merge}(S_j, S_{j+1}) \)

\( j := j + 1 \)

else

\( t_j := t_j + 1 \)

end algorithm

The procedure \( \text{Merge}(S_i, S_k) \) performs a merging of the two sequences \( J_{i1}, \ldots, J_{iS_i} \) and \( J_{j1}, \ldots, J_{jS_k} \), according to weights \( w_i \).

**Theorem 3** The algorithm above finds a minimum \( \sum w_iC_i \) schedule of \((J, P)\).

**Proof:** Initially, \( S_1 \) contains all jobs \( J_i \) with \( r_i = t_1 \). Let \( S_j \) be the set of jobs being currently considered by the algorithm. Suppose \( S_j \) contains all the candidate jobs to be scheduled at time \( t \). That is, \( S_j \) contains all \( J_i \) such that \( r_i \leq t \) and \( J_i \) is still unscheduled.
If \(|S_j| \leq m\), we can schedule all candidates to time \(t\) at this time. The next job available to the system will only be released at time \(t = t_{j+1}\).

If we have more than \(m\) jobs in \(S_j\), we schedule the first “heaviest” jobs in \(S_j\) at time \(t\). If new jobs (not in \(S_j\)) are released at time \(t_j + 1\) (that is, \(t_{j+1} = t_j + 1\)), then the candidate jobs to be scheduled at time \(t = t_j + 1\) will be jobs still in \(S_j\) together with the jobs in \(S_{j+1}\). Else the only candidates to be scheduled at this time, will be the jobs in \(S_j\).

Let \(S\) be the schedule found by the algorithm. Suppose by contradiction that \(\sum w_iC_i(S)\) is not minimum. Then we must have another schedule \(R\) such that \(\sum w_iC_i(R) < \sum w_iC_i(S)\). Let \(J_p\) and \(J_q\), \(p \neq q\), be two earliest jobs such that \(C_p(R) = C_q(S)\), \(C_p(S) \neq C_q(R)\) and \(w_q \neq w_p\) (thus we have \(C_p(S) > C_p(R)\) and \(C_q(R) > C_q(S)\)). If no such jobs \(J_p\) and \(J_q\) exists, then both \(S\) and \(R\) are minimum schedules and the theorem is proved. Else according to the algorithm, \(w_q > w_p\), as \(J_p\) and \(J_q\) are both candidate jobs to be scheduled by the algorithm at time \(t = s_q\). Thus, we can swap \(J_p\) with \(J_q\) in \(R\) (preserving the feasibility of the schedule) and \(\sum w_iC_i(R)\) was not minimum : a contradiction. \(\square\)

**Theorem 4** The algorithm \(\sum w_iC_i\) requires \(O(n(\log n + n/m))\) time.

**Proof** : Ordering \(J\) according to the values of \(r_i\) and ordering all \(S_j\) according to the values of \(w_i\) takes \(O(n \log n)\) time. Scheduling at most \(m\) jobs at a given time \(t\) and updating \(S_j\) can be done in \(O(m)\) time for each iteration of the while loop. The number of iterations is bounded by \(O(n)\), as at each iteration at least one job is scheduled.

Consider the time consumed by the mergings performed by the algorithm. The worst case input, with respect to this step of the algorithm, is when \(t_i = t_{i+1}, 1 \leq i < k\). In this case, we could have the following worst case sequence of number of elements to be merged (we always select \(m\) jobs to be scheduled before performing a merging) :

\[
|S_1| + |S_2| - m \\
|S_1| + |S_2| + |S_3| - 2m \\
\vdots \\
|S_1| + |S_2| + \ldots + |S_k| - (k - 1)m
\]

As \((k - 1)m \leq n\), we have \(O(n/m)\) mergings, each of them of complexity \(O(n)\). Thus the algorithm have overall complexity \(O(n(\log n + n/m))\). \(\square\)

**References**


Figure 1: An operation of (back) chain replacement. Each job $J_j$ is represented by the ordered pair $r_j, d_j$. Job $J_j$ is the only tardy job in the schedule.