T U M

I N S T I T U T F Ü R I N F O R M A T I K

# Scheduling Interval Orders in Parallel

Ernst W. Mayr

TUM-I9426
Mai 1994

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

# Scheduling Interval Orders in Parallel

Ernst W. Mayr*

May 29, 1994

**Abstract**

Interval orders are partial orders defined by having interval representations. It is well known that a transitively oriented digraph $G$ is an interval order iff its (undirected) complement $\bar{G}$ is chordal. We investigate parallel algorithms for the following scheduling problem: Given a system consisting of a set $\mathcal{T}$ of $n$ tasks (each requiring unit execution time) and an interval order $\prec$ over $\mathcal{T}$, and given $m$ identical parallel processors, construct an optimal (*i.e.*, minimal length) schedule for $(\mathcal{T}, \prec)$.

Our algorithm is based on a subroutine for computing so-called scheduling distances, *i.e.*, the minimal number of time steps needed to schedule all those tasks succeeding some given task $t$ and preceding some other task $t'$. For a given interval order with $n$ tasks, these scheduling distances can be computed using $n^3$ processors and $O(\log^2 n)$ time on a CREW-PRAM. We then give an incremental version of the scheduling distance algorithm, which can be used to compute the empty slots in an optimal schedule. From these, we derive the optimal schedule, using no more resources than for the initial scheduling distance computation and considerably improving on previous work by Sunder and He.

The algorithm can also be extended to handle task systems which, in addition to interval order precedence constraints, have individual deadlines and/or release times for the tasks. Our algorithm is the first $\mathcal{NC}$-algorithm for this problem. As another application, it also provides $\mathcal{NC}$-algorithms for some graph problems on interval graphs (which are $\mathcal{NP}$-complete in general).

## 1 Introduction

Highly parallel architectures promise to speed up computation considerably and thus make many more problems amenable to solution. To make use of these architectures, however, it is necessary to decompose an algorithm into parallel steps such that the actions in each parallel step are independent of each other, and to assign the activities to the parallel processors for execution. In this paper, we deal with a variant of this latter problem of *scheduling* tasks on a set of parallel processors. Many, if not most, scheduling problems considered in the literature are $\mathcal{NP}$-complete [20, 15]. On the other hand, there are a few scheduling problems for which fast and (processor) efficient parallel solutions have been found. They include task systems with release times and deadlines [3], task systems with arbitrary precedence constraints for two processor architectures [12], and task systems for

*Institut für Informatik, Technische Universität München, 80290 München, Germany, Phone: +49 89 2105 2680, e-mail: mayr@informatik.tu-muenchen.de

an arbitrary number of processors and precedence constraints which are either in-forests or outforests [4], [11]. More recently, Sunder and He [19] have given the first $\mathcal{NC}$-algorithm for scheduling interval ordered task systems. Apart from some rather special cases [3] and in-forests and out-forests [11], interval orders are the only types of precedence constraints for which the *unit execution time* (UET) scheduling problem is known to be solvable by an $\mathcal{NC}$ algorithm for an arbitrary number of (identical) parallel processors.

In this paper, we consider the class of precedence constraints given by *interval orders*. This class is quite rich, and it has received considerable attention in the past [7, 5, 17, 9]. Concerning scheduling, it is known [18] that the UET scheduling problem on interval orders can be solved sequentially in linear time, and that the problem becomes $\mathcal{NP}$-complete if the UET restriction is dropped. The *algorithm* given in [18] is based on *list scheduling* and appears inherently sequential. In fact, the general problem of list scheduling has been shown to be $\mathcal{P}$-complete [11, 19], and it is therefore very unlikely that any substantial parallel speedup can be achieved.

Our main result is an $\mathcal{NC}$-algorithm for the UET scheduling problem on interval orders and an arbitrary number of parallel processors. It runs in $O(\log^2 n)$ time on $n^3$ processors of a CREW-PRAM [6] whereas the previous algorithm of [19] requires the same time bound but $n^5$ processors on the stronger CRCW-PRAM model which allows concurrent writes in addition to concurrent reads of the same memory cell.

Our result is important in two respects. First, it is one of very few cases where an $\mathcal{NC}$-algorithm could be obtained for a scheduling problem where the number of processors is part of the problem instance. And second, it shows that even though list scheduling in general most likely defies efficient parallelization, there are more restricted cases where powerful techniques for parallel algorithms can be effectively used. In fact, the optimal schedule constructed by our algorithm turns out to be the same as the one obtained by the list scheduling algorithm in [18]. Interval orders occur naturally as precedence constraints in manufacturing problems; for another interpretation see also [18].

The remainder of this paper is organized as follows. In section 2, we introduce the basic notations used in the paper, and we state some fundamental properties about interval orders and schedules. The following section describes an algorithm to compute the length of an optimal schedule for an interval order, as well as an incremental version of this algorithm. These two algorithms are then used in section 4 to design a routine for computing an optimal list schedule for interval orders. The last two sections contain applications of our algorithms as well as some concluding remarks and open problems.

## 2 Fundamental Concepts and Notation

An *interval order* $\prec$ over $n$ elements $t_1, \ldots, t_n$ is given by a set of $n$ closed intervals $I_i$, $i = 1, \ldots, n$ on the real line. The $i$-th element is attached to the $i$-th interval. We say that $t_i \prec t_j$ whenever $I_i$ lies completely to the left of $I_j$, *i.e.*, $x \in I_i$ and $y \in I_j$ implies $x < y$. The system of intervals $I_i$ is called an *interval representation* of the partial order $\prec$. The incomparability graph $\bar{G}$ of a partial order $\vec{G}$ is the undirected complement of the partial order: there is an edge between two vertices $x$ and $y$ in $\bar{G}$ whenever $x$ and $y$ are distinct and neither $x \prec y$ nor $y \prec x$ in $\vec{G}$. It is well-known that a partial order $\vec{G}$ is an interval order iff $\bar{G}$ is *chordal* [18], *i.e.*, each cycle of length greater than three has a chord. The incomparability graph of an interval order is also called an *interval graph*. Obviously, interval graphs are chordal. Figure 1 shows an example of an interval order

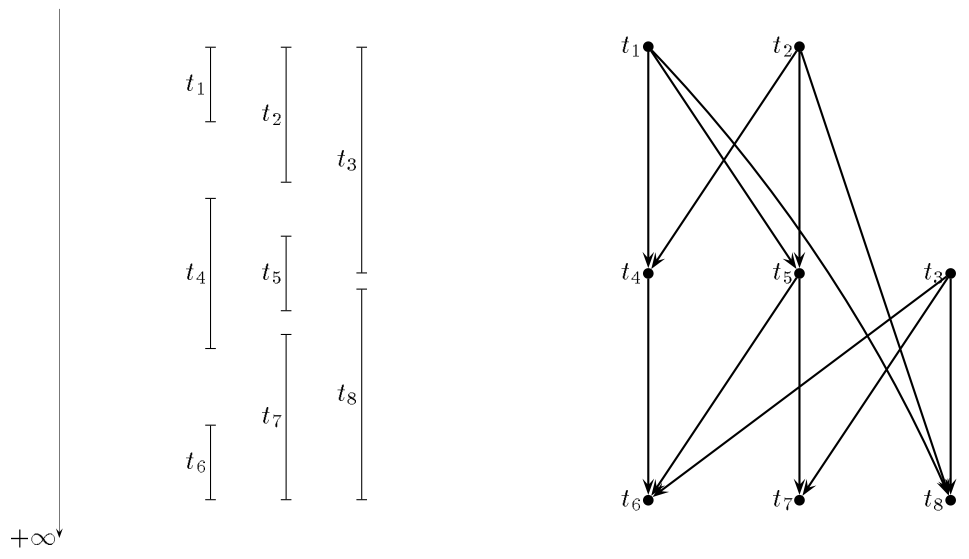

Figure 1: Interval representation and interval order

together with an interval representation for it.

Let $t$ be some element in an interval order. We use $N(t)$ to denote the set of all successors of $t$, *i.e.*,

$$N(t) = \{t';\ t \prec t'\}.$$

Note that $t \notin N(t)$. Analogously, $P(t)$ denotes the set of all predecessors of $t$, $P(t) = \{t';\ t' \prec t\}$. Since $N(t)$ is determined by the right endpoint of $t$'s interval in the interval representation of $\prec$, $N(t)$ and $N(t')$ are comparable wrt. set inclusion for any pair of elements $t$ and $t'$. Using the left endpoints of the intervals instead, the same statement can be seen to hold for the predecessor sets $P(t)$. In the example of Figure 1, we have

$$N_1 = N_2 \supset N_3 \supset N_5 \supset N_4 \supset N_6 = N_7 = N_8\,,$$

$$P_1 = P_2 = P_3 \subset P_4 = P_5 \subset P_8 \subset P_7 \subset P_6\,.$$

Given an interval representation of $\prec$, total orderings of the elements compatible with the ordering of the predecessor sets (resp., successor sets) can easily be obtained in logarithmic time on an $n$-processor EREW-PRAM ($n$ the number of elements in the interval order) using sorting [2, 16].

An *m-processor schedule* for a UET task system $(T, \prec)$, with $T = \{t_1, \ldots, t_n\}$ and $\prec$ a partial order over $T$, is a mapping $S$ of $T$ to the positive integers (timesteps) with the following properties:

1. whenever $t \prec t'$ then $S(t) < S(t')$;

2. at most $m$ tasks are mapped to any one timestep.

The length of the schedule is the largest timestep which has a task mapped to it. An *optimal* schedule is one of minimal length. In terms of continuous time, timestep $i$ comprises the time interval $[i-1, i)$. Note that all tasks require unit time for execution, and that (the size of) the interval associated with a task in an interval representation for the interval order has nothing to do with its execution time.

A *list schedule* for a scheduling problem is obtained as follows: Let $m$ be the number of parallel processors. The tasks are arranged in a linear list, according to some criterion. The tasks are then scheduled, timestep for timestep, from this list according to the following rule:

> Take the first $m$ tasks in the list which are *executable*, *i.e.*, for which all of their predecessors have already been assigned to earlier steps, remove them from the list and assign them one per processor. If only $m' < m$ such tasks are available, $m - m'$ processors remain idle for this timestep (we also say that there are $m - m'$ *empty slots* in this timestep. Note that the tasks selected for execution in a timestep need not be in contiguous positions in the list.

The parallel complexity of list scheduling has been studied in [11] and [19] and shown to be $\mathcal{P}$-complete [13] for the following general cases:

1. no precedence constraints, arbitrary integer execution times, any fixed number $m > 1$ of parallel processors [11];
2. arbitrary precedence constraints, unit execution time, number of parallel processors part of the problem instance [19].

Papadimitriou and Yannakakis [18] have shown that, for task systems with an interval order as precedence constraints, the list schedule obtained from any list in which the tasks are arranged in order of nonincreasing successor set is optimal. While the list scheduling paradigm appears inherently sequential as argued above, it actually turns out that list schedules can be computed fast in parallel in the case of interval orders, albeit by a rather different algorithmic approach which just happens to produce an optimal schedule identical to the list schedule of [18].

## 3 Scheduling Distance Computation

We first study some properties of the (optimal) schedules obtained by the list scheduling algorithm of [18]. For the time being, all we assume about the list of tasks is that, whenever $t'$ follows $t$ in this list, then $N(t') \subseteq N(t)$. We start by showing that there are subsets of tasks which have properties very similar to the blocks studied by Coffman and Graham in [1] in their algorithm for the 2-processor scheduling problem.

**Theorem 1** *Let $(T, \prec)$ be an interval order, and let opt be the minimal length of a schedule for $(T, \prec)$ on $m$ processors. Then there exist* pairwise disjoint *subsets (called* blocks*) $\chi_i \subseteq T$, $i = 1, \dots, r$ (for some $r$) with*

*1.* $t \in \chi_i, t' \in \chi_{i-1} \quad \Rightarrow \quad t \prec t'$,

*2.* $|\chi_i| \equiv 1 \pmod{m}$,

*3.* $\sum_{i=1}^{r} \left\lceil \frac{|\chi_i|}{m} \right\rceil = opt$.

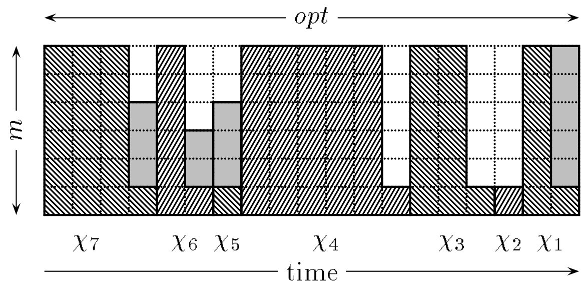

Figure 2: Blocks in a schedule

**Proof:** Let $S$ be the list schedule obtained by the list scheduling algorithm in [18]. Then $S$ is optimal, *i.e.,* of length *opt*. Let $\tau_0$ be the number of the last timestep of $S$, and let $T_0$ be the tasks scheduled in timestep $\tau_0$. Also, let $\tilde{t}_0$ be any task in $T_0$. Now define inductively, as long as $\tau_{i-1} > 0$:

$\tau_i$ := the latest timestep before $\tau_{i-1}$ s.t.
(i) timestep $\tau_i$ contains an empty slot, or
(ii) there is a task $t'$ scheduled in timestep $\tau_i$ with $N(t') \subset N(\tilde{t}_{i-1})$
(by default, $\tau_i := 0$ if there is no such time step);
$T_i$ := the set of tasks scheduled in timestep $\tau_i$;
$\tilde{t}_i$ := some task in $T_i$ with maximal successor set;
$\chi_i$ := $\{\tilde{t}_{i-1}\} \cup \{$tasks in timesteps strictly between $\tau_i$ and $\tau_{i-1}\}$.

Let $\chi_r, \ldots, \chi_1$ be the blocks constructed by this algorithm, let $i > 1$ and assume $t \in \chi_i$ and $t' \in \chi_{i-1}$. By construction, $N(t) \supseteq N(\tilde{t}_i)$. If timestep $\tau_i$ contains an empty slot, then each task in timesteps later than $\tau_i$ must be a successor of some task in $T_i$. Since the successor sets of the tasks are pairwise comparable wrt. set inclusion, and since $\tilde{t}_i$ has a maximal successor set of all the tasks scheduled in timestep $\tau_i$, $\tilde{t}_i$ precedes, in this case, all tasks in later timesteps. If, on the other hand, there is a task $t''$ scheduled in timestep $\tau_i$ with $N(t'') \subset N(\tilde{t}_{i-1})$, then every task in $\chi_{i-1}$ must have a predecessor scheduled in timestep $\tau_i$, because the tasks in $\chi_{i-1}$ appear in the list before $t''$, at least one of them would be executable and would be scheduled by the list scheduling routine alongside with $\tilde{t}_i$, instead of $t''$. Again, because $\tilde{t}_i$ has a maximal successor set in its timestep, $\tilde{t}_i$ precedes all tasks in $\chi_{i-1}$, in particular $t'$. In either case, since $N(t) \supseteq N(\tilde{t}_i)$, we conclude that $t \prec t'$, proving (1). Claim (2) is clear from the construction of the blocks $\chi_i$. From (1) it follows that, in any schedule, all tasks in $\chi_i$ must be scheduled in earlier timesteps than any task in $\chi_{i-1}$. Since there are $m$ parallel processors available, the tasks in $\chi_i$ require at least $\left\lceil \frac{|\chi_i|}{m} \right\rceil$ many timesteps, and hence every schedule has length at least

$$\sum_{i=1}^{r} \left\lceil \frac{|\chi_i|}{m} \right\rceil ,$$

while, by construction, the length of $S$ is equal to this quantity, implying the optimality of $S$ (giving another, succinct proof of the main result in [18]) and establishing (3). □

Figure 2 shows an example of blocks in a schedule. In this figure, each row corresponds to the timesteps on one of the $m$ processors, and tasks are shown as squares. Tasks belonging to blocks are hatched (for clarity at different angles in adjacent blocks), and tasks not in blocks are shown in grey. Empty slots are left white.

**Definition 1** *Let $t$ and $t'$ be two tasks of some task system $(T, \prec)$. Then $I(t,t') := N(t) \cap P(t')$ is the set of all those tasks that are simultaneously successors of $t$ and predecessors of $t'$, and $\prec_{I(t,t')}$ is the partial order $\prec$ restricted to $I(t,t')$.*

If $t$ and $t'$ are incomparable wrt. $\prec$, then $I(t,t') = \emptyset$. It should also be clear that, if $\prec$ is an interval order on $T$, then so is $\prec_{I(t,t')}$ on $I(t,t')$, for any $t$ and $t'$.

**Definition 2** *Let $t$ and $t'$ be two tasks of some task system $(T, \prec)$. Then the* scheduling distance *$D(t,t')$ is the length of an optimal schedule for $(I(t,t'), \prec_{I(t,t')})$.*

$D(t,t')$ is the minimal number of steps required to schedule all tasks that are successors of $t$ and predecessors of $t'$. Clearly, if $t$ and $t'$ are incomparable wrt. $\prec$, then $D(t,t') = 0$.

**Lemma 1** *Let $(T, \prec)$ be some task system, let $t, t' \in T$, $U \subseteq I(t,t')$, $U \neq \emptyset$, and let $j, k \in \mathbb{N}$ such that for all $t'' \in U$*

1. *$D(t,t'') \geq j$, and*
2. *$D(t'',t') \geq k$.*

*Then*

$$D(t,t') \geq j + k + \left\lceil \frac{|U|}{m} \right\rceil .$$

**Proof:** In any schedule for $I(t,t')$, there are, by the definition of scheduling distance, at least $j$ timesteps before the first task of $U$ can be scheduled, and there are at least $k$ steps after the last timestep for a task in $U$. Since the tasks in $U$ require at least $\left\lceil \frac{|U|}{m} \right\rceil$ timesteps themselves, the claim follows. □

**Lemma 2** *Let $t, t', \bar{t}$ and $\bar{t}'$ be tasks in some task system $(T, \prec)$ such that $I(t,t') \subseteq I(\bar{t}, \bar{t}')$. Then $D(t,t') \leq D(\bar{t}, \bar{t}')$.*

**Proof:** This follows immediately since $(I(\bar{t},\bar{t}'), \prec_{I(\bar{t},\bar{t}')})$ contains $(I(t,t'), \prec_{I(t,t')})$. □

Note that, in the above lemma, $t$ and $\bar{t}$ (resp., $t'$ and $\bar{t}'$) need not be comparable wrt. $\prec$, even though $I(t,t') \subseteq I(\bar{t}, \bar{t}')$.

**Lemma 3** *Let $(T, \prec)$ be a task system, with $\prec$ an interval order, let $t, t' \in T$ and $U \subseteq I(t,t')$, $U \neq \emptyset$. Further, let $t_1 \in U$ be a task with minimal predecessor set, and let $t_2 \in U$ be a task with minimal successor set. Then*

$$D(t,t') \geq D(t,t_1) + \left\lceil \frac{|U|}{m} \right\rceil + D(t_2,t') .$$

**Proof:** Since $\prec$ is an interval order, the successor sets $N(\cdot)$ are pairwise comparable wrt. set inclusion, as are the predecessor sets $P(\cdot)$. Therefore, $P(t_1) \subseteq P(t'')$ and $N(t_2) \subseteq N(t'')$ for all $t'' \in U$. Hence, by the previous lemma, $D(t,t_1) \leq D(t,t'')$ and $D(t_2,t') \leq D(t'',t')$ for all $t'' \in U$, and thus the claim follows by applying Lemma 1. □

We now assume in the following wlog that the set of tasks $T = \{t_1, \ldots, t_n\}$, and that the tasks are renamed such that $N(t_1) \supseteq N(t_2) \supseteq \ldots \supseteq N(t_n)$, *i.e.,* in order of nonincreasing (size of) successor set. We also let $\pi$ be a permutation of $\{1, \ldots, n\}$ such that $P(t_{\pi(i)}) \subseteq P(t_{\pi(i+1)})$ for all $i = 1, \ldots, n-1$, *i.e.,* $t_{\pi(1)}, t_{\pi(2)}, \ldots, t_{\pi(n)}$ is in order of nondecreasing (size of) predecessor set. Finally, we let $(t_{\pi(i),j})_{1 \le j \le n-i+1}$ be the subsequence of $(t_1, t_2, \ldots, t_n)$ consisting of those $t_k$ with $\pi^{-1}(k) \ge i$, *i.e.,* $(t_{\pi(i),1}, t_{\pi(i),2}, \ldots, t_{\pi(i),n-i+1})$ is $(t_{\pi(i)}, t_{\pi(i+1)}, \ldots, t_{\pi(n)})$ rearranged in order of nonincreasing successor set. It is clear that these sequences as well as $\pi$ and its inverse can be computed using sorting within time $O(\log n)$ on $n^2$ processors.

Given a task system $(T, \prec)$, with $|T| = n$ tasks and an interval order $\prec$, the following algorithm computes the scheduling distances for all pairs $(t, t') \in T^2$:

```
algorithm $sd(T, \prec, m)$:
    determine the permutation $\pi$ and its inverse;
    determine, for all $i = 1, \ldots, n$, the sequences $(t_{\pi(i),j})_{1 \le j \le n-i+1}$;
    forall $i, j$ do in parallel
      $D_0(t_{\pi(i)}, t_{\pi(j)}) := \left\lceil \frac{|I(t_{\pi(i)}, t_{\pi(j)})|}{m} \right\rceil$;
    for $p := 1$ to $\lceil \log n \rceil$ do
      forall $i, j$ with $t_{\pi(i)} \prec t_{\pi(j)}$ do in parallel
        $d'(i,j) := \max\{\left\lceil \frac{k}{m} \right\rceil + D_{p-1}(t_{\pi(i),k}, t_{\pi(j)});\ k$ such that $t_{\pi(i),k} \prec t_{\pi(j)}\}$;
      forall $i, j$ with $t_{\pi(i)} \prec t_{\pi(j)}$ do in parallel
        $D_p(t_{\pi(i)}, t_{\pi(j)}) := \max\{D_{p-1}(t_{\pi(i)}, t_{\pi(j)}), D_{p-1}(t_{\pi(i)}, t_{\pi(k)}) + d'(k,j);$
                                   $k$ such that $t_{\pi(i)} \prec t_{\pi(k)} \prec t_{\pi(j)}\}$;
    return($D_{\lceil \log n \rceil}(\cdot, \cdot)$)
```

**Lemma 4** *Algorithm sd correctly computes the scheduling distances.*

**Proof:** The two forall $i, j$ loops together compute

```
      forall $i, j$ with $t_{\pi(i)} \prec t_{\pi(j)}$ do in parallel
        $D_p(t_{\pi(i)}, t_{\pi(j)}) := \max\{D_{p-1}(t_{\pi(i)}, t_{\pi(j)}),$
                                   $D_{p-1}(t_{\pi(i)}, t_{\pi(k)}) + \left\lceil \frac{l}{m} \right\rceil + D_{p-1}(t_{\pi(k),l}, t_{\pi(j)});$
                                   $k$ such that $t_{\pi(i)} \prec t_{\pi(k)} \prec t_{\pi(j)}$ and
                                   $l$ such that $t_{\pi(k),l} \prec t_{\pi(j)}\}$
```

Using Lemma 3, a simple induction now shows that, for all $t$ and $t'$ and for all $p$, we always have $D_p(t, t') \le D(t, t')$, *i.e.,* the $D_p(t, t')$ are always on the conservative side.

On the other hand, if $I(t, t')$ is empty or contains just one block, then clearly, by the initialization, $D_0(t, t') = D(t, t')$. If $I(t, t')$ contains $q$ blocks, with $2^p \le q \le 2^{p+1} - 1$, consider the block in the "middle", which is the one with number $q'$, $q' = \left\lceil \frac{q+1}{2} \right\rceil$. Let $t_1$ (resp., $t_2$) be any task in this block which has no predecessor (resp., successor) within the block. Then $I(t, t_1)$ and $I(t_2, t')$ both have strictly fewer than $2^p$ blocks, and we may assume by induction that $D_{p-1}(t, t_1) = D(t, t_1)$ and $D_{p-1}(t_2, t') = D(t_2, t')$. Since the monotonicity property (Lemma 2) also holds for all $D_p$ instead of $D$, we have that for all tasks $t''$ in the block numbered $q'$ $D_{p-1}(t, t_1) \le D_{p-1}(t, t'')$ and $D_{p-1}(t_2, t') \le D_{p-1}(t'', t')$. Hence it follows from the algorithm that $D_p(t, t') = D(t, t')$. □

We have now everything in place for

**Theorem 2** *Let $(T, \prec)$ be a task system, with $|T| = n$ tasks, each requiring unit time for execution, and $\prec$ an interval order, and let $m$ be an integer $\geq 1$. Then the length of an optimal $m$-processor schedule for $(T, \prec)$ can be computed in time $O(\log^2 n)$ on a CREW-PRAM with $n^3$ processors.*

**Proof:** We add to $(T, \prec)$ a task $t_{top}$ preceding, and a task $t_{bot}$ succeeding all tasks in $T$. This can clearly be done in such a way that the resulting partial order is still an interval order. We then use algorithm *sd* with parameters $T, \prec$, and $m$ in order to compute $D(t_{top}, t_{bot})$, which equals the minimal number of steps needed to execute $(T, \prec)$ on $m$ processors, and which is therefore the optimal schedule length. The bounds for the running time and the number of CREW-PRAM processors follow immediately from the description of the algorithm. □

While we have just constructed a (moderately) efficient $\mathcal{NC}$-algorithm to compute the *length* of an optimal schedule for a task system whose precedence constraints form an interval order, we are still faced with the problem to compute such a schedule itself, *i.e.,* the actual assignment of the tasks to timesteps. We are going to conclude this section presenting an *incremental* version of the scheduling distance algorithm which, as will be shown in the next section, can be used to actually compute an optimal schedule.

For this, let $(T, \prec)$ be a task system as above, with $\prec$ being an interval order, let $\tilde{t}$ be some task in $T$, and let $a$ be some nonnegative integer. From the given task system, we construct a new task system $(T', \prec')$ as follows: we add to $T$ $a$ new tasks which will all be identical and named $\hat{t}$. Each $\hat{t}$ has no successor in $(T', \prec')$, and its predecessors are exactly those $t \in T$ with $N(t) \supseteq N(\tilde{t})$. It should be clear that $\prec'$ is still an interval order. We denote the resulting task system by $(T, \prec, \tilde{t}, a)$.

To compute the optimal schedule length for $(T, \prec, \tilde{t}, a)$, we can proceed as before. We can add tasks $t_{top}$ and $t_{bot}$ in such a way that $t_{top}$ (resp., $t_{bot}$) precedes (resp., succeeds) every other task, and then apply algorithm *sd*. However, if we already know the scheduling distances for $(T, \prec)$ augmented by $t_{top}$ and $t_{bot}$ (actually, we will not even use any precomputed scheduling distances to $t_{bot}$), the following algorithm provides a more efficient way to obtain the optimal schedule length for $(T, \prec, \tilde{t}, a)$. We use $n$ to denote $|T|$.

**algorithm** *isd*$(T, \prec, \tilde{t}, a, m)$:

```
    co compute $D(t_{top}, \hat{t})$; since the $a$ copies of $\hat{t}$ are identical, this needs to be done
        for just one of them oc
    sort $I(t_{top}, \hat{t})$ in order of nondecreasing scheduling distance from $t_{top}$;
    let $(t_{i_j})_{1 \leq j \leq n'}$ be the resulting sequence;
    $D(t_{top}, \hat{t}) := \max\{0, D(t_{top}, t_{i_j}) + \left\lceil \frac{n'-j+1}{m} \right\rceil ;\ 1 \leq j \leq n'\}$;
    co compute $D(t_{top}, t_{bot})$ oc
    sort $I(t_{top}, t_{bot})$ in order of nondecreasing scheduling distance from $t_{top}$;
    let $(t_{i_j})_{1 \leq j \leq n+a}$ be the resulting sequence;
    $D(t_{top}, t_{bot}) := \max\{0, D(t_{top}, t_{i_j}) + \left\lceil \frac{n+a-j+1}{m} \right\rceil ;\ 1 \leq j \leq n+a\}$;
    return $D(t_{top}, t_{bot})$
```

**Lemma 5** *Algorithm isd correctly computes the optimal schedule length for the extended task system $(T, \prec, \tilde{t}, a)$.*

**Proof:** Consider first some optimal schedule for $I(t_{top}, \hat{t})$ in the extended task system. Then the scheduling distances from $t_{top}$ to all tasks in the (wrt. time) last block of this schedule are known by assumption. Hence, the first loop in algorithm *isd* is guaranteed to also pick the minimal value of these, resulting in computing the correct value for $D(t_{top}, \hat{t})$.

As for $D(t_{top}, t_{bot})$, also consider some optimal schedule for $(T, \prec, \tilde{t}, a)$. Then, if the $a$ copies of $\hat{t}$ occur at all within blocks, they must all be contained in the (wrt. time) last block. As before, this implies that the algorithm computes the correct scheduling distance. □

With the parameters as given above, algorithm *isd* requires time $O(\log n)$ and $n$ processors on a CREW-PRAM. By using only one copy of $\hat{t}$ and observing in the computations that it actually represents $a$ identical copies, these resource bounds hold independently of $a$.

## 4 Constructing an Optimal Schedule

With the scheduling distance and the incremental scheduling distance algorithms at hand, we can now compute the actual assignment of tasks to timesteps in an optimal schedule for interval ordered task systems. Let $(T, \prec)$ be the given task system, with $\prec$ an interval order. Sort $T$ in order of nonincreasing (size of) successor set and partition the resulting list into maximal *segments* consisting of tasks with identical successor sets. Let $s$ be the number of segments obtained. Within each segment, we sort the tasks according to non-decreasing (size of) predecessor set. In what follows, we are going to compute the schedule obtained by applying the list scheduling algorithm of [18] to the resulting list. By the results in [18] we know that the schedule we obtain is optimal.

We first add to $(T, \prec)$ (maintaining interval orderedness) a task $t_{top}$ preceding all tasks in $T$, and, for each $k = 1, \ldots, s$, a task $t_{k,bot}$ succeeding $t_{top}$ and all tasks in the first $k$ segments and having no successors itself. We also set $t_{bot} := t_{s,bot}$, and we call the extended system $(T'', \prec'')$.

**Definition 3** *For $k = 1, \ldots, s$, $G_k$ is the subsystem of $(T'', \prec'')$ given by the tasks $t_{top}$ and $t_{k,bot}$ and all tasks in $I(t_{top}, t_{k,bot})$,* i.e., *all tasks between $t_{top}$ and $t_{k,bot}$.*

Using algorithm *sd*, we can compute the scheduling distances for $(T'', \prec'')$ in time $O(\log^2 n)$ on a CREW-PRAM with $n^3$ processors, where $n$ is the cardinality of $T$.

**Definition 4** *We say that segment $i$ jumps to segment $j > i$ with multiplicity $b$ if, in the list schedule, the last time step containing a task from segment $i$ and no task from any earlier segment also contains (exactly) $b$ tasks from segment $j$.*

It should be clear that once we know all the jumps (with their multiplicities), it is straightforward using prefix computations to determine the (optimal) list schedule: we first remove from every segment (in order of nondecreasing predecessor set) as many tasks as are jumped to, and add these tasks to the end of those segments from which the jumps occur. Then we group every segment into chunks of size $m$ (the last chunk for each segment can of course be smaller). The $i$th chunk then, in order, corresponds to the set of tasks scheduled in timestep $i$.

We thus now concentrate on computing the jumps and their multiplicities.

**Definition 5** *Let $G_{k,j,a}$, for $1 \leq j < k$ and $0 \leq a \leq s \cdot (m-1)$, be $G_k$ augmented by $c$ identical new tasks which have all tasks in the first $j-1$ segments as predecessors and $t_{k,bot}$ as the only successor (preserving interval orderedness).*

Given the scheduling distances for the $G_k$, which we computed above, the scheduling distance $D(t_{top}, t_{k,bot})$ in $G_{k,j,c}$ can be computed in time $O(\log n)$ using $n$ processors, by applying algorithm *isd*. In the following, we use, for a task system $G$ including tasks $t_{top}$ and $t_{bot}$ as above (*i.e.,* $t_{top}$ precedes, and $t_{bot}$ succeeds every other task), *opt*$(G)$ to denote the optimal length of a schedule for $I(t_{top}, t_{bot})$ in $G$ (which is equal to the scheduling distance $D(t_{top}, t_{bot})$ in $G$).

```
algorithm list_schedule$(T, \prec)$:
    compute the scheduling distances for $(T'', \prec'')$;
    forall applicable $k$ and $j$ do in parallel
      use binary search to determine the maximal $c$ such that $opt(G_{k,j,c}) = opt(G_k)$;
      co this $c =: c_{k,j}$ is the number of empty slots in the list schedule for the first $k$
          segments after all tasks in the first $j-1$ segments have been executed oc
      $\tilde{c}_{k,j} := c_{k,j} - c_{k,j+1}$;
      co $\tilde{c}_{k,j}$ is the number of empty slots in the list schedule for the first $k$ segments
          in the last timestep with a task from segment $j$ and no task from any
          earlier segment oc
      $jump(j,k) := \tilde{c}_{k-1,j} - \tilde{c}_{k,j}$;
    co $jump(k,j)$ is the exact multiplicity of the jump from segment $j$ to segment $k$ oc.
```

Algorithm *list_schedule* still requires $n^3$ processors and time $O(\log^2 n)$ on a CREW-PRAM.

## 5 Applications and Extensions

Our algorithm for scheduling task systems with interval orders as precedence constraints can be extended to also handle, in addition, individual release times and deadlines for the tasks. We obtain an algorithm using $O(\log^4 n)$ time on $n^8$ processors of a CREW-PRAM. While this is the first $\mathcal{NC}$-algorithm for this problem, it is too involved to be presented here and is deferred to another, forthcoming paper.

The parallel interval order scheduling algorithm presented in the previous sections can also be used to show that certain decision problems, while they are $\mathcal{NP}$-complete in general, can be solved in $\mathcal{NC}$ for interval graphs.

Interval graphs are undirected graphs, defined as the *incomparability graphs* of interval orders, *i.e.,* the undirected complements of interval orders. We assume in the following that interval graphs are given by an interval representation (as for a corresponding interval order). If this is not the case, there are $\mathcal{NC}$-algorithms (based on transitive orientation) known to compute an interval representation [14, 10], given a standard adjacency list or adjacency matrix representation of the interval graph.

We first show

**Lemma 6** *Let $G$ be some interval graph, which is the incomparability graph of some interval order $\vec{G} = (T, \prec)$. Let $C_1, C_2 \subseteq T$ be two cliques in $G$, and assume that $t \in C_1$, $t' \in C_2$, and $t \prec t'$ in $\vec{G}$. Then there are no $t_1 \in C_1$ and $t_2 \in C_2$ such that $t_2 \prec t_1$ in $\vec{G}$.*

**Proof:** Assume to the contrary that there are $t_1 \in C_1$ and $t_2 \in C_2$ such that $t_2 \prec t_1$ in $\vec{G}$. If $N(t) \subseteq N(t_2)$ then $t' \in N(t)$ implies that $t_2 \prec t'$ which is impossible. Otherwise, if $N(t) \supseteq N(t_2)$ then $t \prec t_1$ which again is a contradiction. □

Lemma 6 implies that, if there is a partition of $T$ into cliques in the interval graph $G$ such that none of the cliques contains more than $m$ vertices then there is an $m$-processor schedule for the task system $\vec{G}$ of length equal to the number of cliques in the partition.

**Definition 6** *For the following definitions, also see [8].*
*1. The problem PARTITION INTO CLIQUES is: Given an undirected graph $G$ and some nonnegative integer $k$, can the vertex set of $G$ be partitioned into at most $k$ cliques?*
*2. The problem PARTITION INTO TRIANGLES is: Given an undirected graph $G$, can the vertex set of $G$ be partitioned in such a way that each class in the partition induces a triangle in $G$?*

Both of these problems are well known to be $\mathcal{NP}$-complete for general graphs. From the discussion above, we immediately obtain

**Lemma 7** *The problem PARTITION INTO CLIQUES is in $\mathcal{NC}$ when restricted to interval graphs.*

**Proof:** Let $(G, k)$ be the instance given, and let $\vec{G}$ be an interval order with $G$ as incomparability graph. Also, let $n$ be the number of vertices in $G$. Then, using Lemma 6, there is an $n$-processor schedule for $\vec{G}$ of length at most $k$ iff the vertex set of $G$ can be partitioned into at most $k$ cliques. □

**Lemma 8** *The problem PARTITION INTO TRIANGLES is in $\mathcal{NC}$ when restricted to interval graphs.*

**Proof:** Let $G$ be the instance given, and let $\vec{G}$ be an interval order with $G$ as incomparability graph. Then there is a 3-processor schedule for $\vec{G}$ without any empty slot iff the vertex set of $G$ can be partitioned into triangles. □

## 6 Conclusion and Open Problems

We have presented in this paper a parallel algorithm for optimally scheduling unit execution time task systems, whose precedence constraints form an interval order, on $m$ identical parallel processors, where $m$ is part of the problem instance. This algorithm runs in time $O(\log^2 n)$, $n$ the number of tasks in the task system, and requires $n^3$ processors of a CREW-PRAM. Compared to the only previous parallel algorithm for this problem, which runs within the same asymptotic time bound, but requires $n^5$ processors on the much more powerful CRCW-PRAM model, our algorithm represents a major improvement.

It is also important since, other than trees (and empty precedence constraints), interval orders are the only case where $\mathcal{NC}$-algorithms are known even when the number $m$ of target processors is arbitrary and part of the problem instance.

In terms of the work performed by our algorithm, there is of course still a huge gap when compared with the linear time sequential algorithm given in [18]. It is an intriguing open problem to find whether the number of processors (and maybe even the time) used by our algorithm can be reduced, assuming even that the transitive closure of the partial order representing the precedence constraints is given to the algorithm in some convenient form. It is also an interesting question to see whether the scheduling distance computation for (basically) all pairs of tasks in the task system can be avoided, and whether there is a more direct approach to determine an optimal schedule.

List scheduling in general is known to be $\mathcal{P}$-complete. As it turns out, the pairwise comparability of the successor resp. predecessor set in an interval order still makes it possible, to compute optimal list schedules for interval orders in $\mathcal{NC}$. One can ask whether there are other restrictions for partial orders allowing similar results.

## References

[1] E. Coffman, Jr. and R. Graham. Optimal scheduling for two processor systems. *Acta Inf.*, 1:200–213, 1972.

[2] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, 1988.

[3] E. Dekel and S. Sahni. Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput.*, C-32(3):307–315, March 1983.

[4] D. Dolev, E. Upfal, and M. Warmuth. Scheduling trees in parallel. In P. Bertolazzi and F. Luccio, editors, *VLSI: Algorithms and Architectures. Proceedings of the International Workshop on Parallel Computing and VLSI (Amalfi, Italy, May 23-25, 1984)*, pages 91–102, Amsterdam-New York-Oxford, 1985. Elsevier North-Holland.

[5] P. C. Fishburn. *Interval orders and interval graphs. A study of partially ordered sets.* John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1985. Wiley Interscience Series in Discrete Mathematics.

[6] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA)*, pages 114–118, New York, 1978. ACM, ACM Press.

[7] D. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15:835–855, 1965.

[8] M. R. Garey and D. S. Johnson. *Computers and intractability. A guide to the theory of $\mathcal{NP}$-completeness.* W.H. Freeman and Company, New York-San Francisco, 1979.

[9] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1:180–187, 1972.

[10] D. Helmbold and E. Mayr. Applications of parallel scheduling to perfect graphs. In G. Tinhofer and G. Schmidt, editors, *Proceedings of the International Workshop WG '86, Bernried, FRG, June 1986. Graph-Theoretic Concepts in Computer Science.*, LNCS 246, pages 188–203, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1987. Springer-Verlag.

[11] D. Helmbold and E. Mayr. Fast scheduling algorithms on parallel computers. In F. P. Preparata, editor, *Advances in Computing Research; Parallel and Distributed Computing*, volume 4, pages 39–68. JAI Press Inc., Greenwich, CT-London, 1987.

[12] D. Helmbold and E. W. Mayr. Two processor scheduling is in $\mathcal{NC}$. *SIAM J. Comput.*, 16(4):747–759, August 1987.

[13] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume A: Algorithms and complexity*, pages 67–161. Elsevier North-Holland, Amsterdam-New York-Oxford, The MIT Press: Cambridge, MA-London, 1990.

[14] D. Kozen, U. V. Vazirani, and V. V. Vazirani. $\mathcal{NC}$ algorithms for comparability graphs, interval graphs and testing for unique perfect matching. In *Proceedings Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pages 496–503, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1985. Springer-Verlag.

[15] B. Lageweg, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Computer aided complexity classification of combinatorial problems. *Commun. ACM*, 25:817–822, 1982.

[16] T. Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the 16th Ann. ACM Symposium on Theory of Computing (Washington, D.C.)*, pages 71–80, New York, 1984. ACM, ACM Press.

[17] J. Naor, M. Naor, and A. A. Schäffer. Fast parallel algorithms for chordal graphs. *SIAM J. Comput.*, 18(2):327–349, April 1989.

[18] C. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Comput.*, 8(3):405–409, August 1979.

[19] S. Sunder and X. He. Scheduling interval ordered tasks in parallel. In P. Enjalbert, A. Finkel, and K. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science, STACS 93 (Würzburg, Germany, February 1993)*, LNCS 665, pages 100–109, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong-Barcelona-Budapest, 1993. GI, afcet, Springer-Verlag.

[20] J. Ullman. $\mathcal{NP}$-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.