June 1986 Report No. STAN-CS-86-1118

Applications of Parallel Scheduling to Perfect Graphs

by

David Helmbold and Ernst Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305

Applications of Parallel Scheduling to Perfect Graphs

David Helmbold and Ernst Mayr
Stanford University

Abstract

We combine a parallel algorithm for the two processor scheduling problem, which runs in polylog time on a polynomial number of processors, with an algorithm to find transitive orientations of graphs where they exist. Both algorithms together solve the maximum clique problem and the minimum coloring problem for comparability graphs, and the maximum matching problem for co-comparability graphs. These parallel algorithms can also be used to identify permutation graphs and interval graphs, important subclasses of perfect graphs.

This work was supported by in part by a grant from the AT&T Foundation, ONR contract N00014-85-C-0731, and NSF grant DCR-8351757.

1 Introduction

We present several parallel algorithms for graph problems, in particular for perfect graphs. Our main result is a deterministic $\mathcal{NC}$ algorithm for solving the two processor unit execution time scheduling problem, answering an important open problem posed in [24]. We also present an $\mathcal{NC}$ algorithm for transitively orienting comparability graphs. By combining these two results, we obtain an $\mathcal{NC}$ algorithm for the matching problem on co-comparability graphs (the complements of comparability graphs) and nearly co-comparability graphs. In addition our transitive orientation algorithm gives us $\mathcal{NC}$ algorithms for several additional problems, such as identifying permutation graphs and finding the maximum weighted clique and optimal colorings in comparability graphs. Comparability, co-comparability, and permutation graphs are all important subclasses of perfect graphs.

The most fundamental scheduling problems involve unit time execution tasks with precedence constraints restricting the order of execution. When the number of processors varies, the scheduling problem is &?-complete [23]. There are no published polynomial time algorithms for a fixed number of processors greater than two. The first polynomial time algorithm for the two processor case was published in 1969 [5]. Faster algorithms for the same problem were given by Coffman and Graham [2], and later, Gabow [6,7] found an asymptotically optimal algorithm. Recently, Vazirani and Vazirani have published a randomized parallel solution [24]. Like Fujii et al. they use the connection between matching and two processor scheduling, so their algorithm relies on an $\mathcal{RNC}$ matching subroutine such as [14] or [18].

In contrast, our scheduling algorithm [12] is deterministic and does not require the aid of a matching subroutine. Therefore we are able to exploit the relationship between matching and two processor scheduling in the other direction, obtaining a, deterministic parallel maximum matching algorithm for co-comparability graphs.

The only ingredient required to convert our scheduling algorithm into a matching result is an $\mathcal{NC}$ transitive orientation subroutine. This routine takes an undirected graph and directs the edges so that the resulting digraph is transitively closed. Those graphs which can be transitively oriented are called comparability graphs. The complements of comparability graphs are co-comparability graphs. Kozen, Vazirani and Vazirani, in independent work, coupled a different transitive orientation routine with our two processor scheduling algorithm to achieve an $\mathcal{NC}$ matching algorithm on co-comparability graphs [16]. Our transitive orientation subroutine is also the key element in our algorithms for testing for permutation graphs and finding maximum weighted cliques and optimal colorings on comparability graphs.

2 Main Theorems and Applications

In this section we give some definitions, state our main results, and prove several important consequences.

As our model of parallel computation, *we* use the *Parallel Random Access Machine* or PRAM as defined in [4]. A PRAM consists of an unbounded number of identical processors running synchronously. Each such processor can be thought of, for the purpose of this paper, as an ordinary RAM [1], with local memory. A PRAM also contains an unbounded number of global memory cells which every processor can access in one timestep. We allow that several processors read the same memory cell simultaneously. However, several processors must not write simultaneously to the same memory cell. Every processor has stored, in one of its registers, its unique processor index. All processors execute the same program. Since the instructions may depend on the processor index, the effect of an instruction will in general vary from processor to processor.

When measuring the complexity of parallel algorithms (for the PRAM model), we are mainly interested in the amount of time an algorithm uses, and the number of processors it employs. Time will be the number of parallel steps taken by the PRAM, and the number of processors will be the highest index of a processor active in the computation.

The class of parallel algorithms running in time which is bounded by a polynomial in the logarithm of the size of the input, and using a number of processors polynomial in the input size, has experienced considerable interest. One reason is that the algorithms in this class are considered very fast (the "speedup" over their sequential counterparts is exponential), and they use a "reasonable" amount of hardware, i.e. processors. Another reason is that this class is very robust under (reasonable) variations in the definitions of the underlying machine model. The class is commonly referred to as $\mathcal{NC}$, owing to its original definition for the boolean circuit model of parallel computation in [20].

A *perfect graph* is an undirected graph where the chromatic number and maximum clique size of every induced subgraph coincide. A *precedence graph* is an acyclic, transitively closed digraph. We use $\{a, b\}$ to denote an undirected edge, and (a, b) to denote a directed edge or arc from vertex a to vertex b. Thus if arcs (a, b) and (b, c) are in a precedence graph, then so is the arc (a, c). A *comparability graph* is an undirected graph with the property that every edge can be assigned a direction such that the resulting graph is a precedence graph. The complement of a comparability graph is a *co-comparability* graph. Precedence graphs are equivalent to partial orders. Some graphs, such as a simple three-cycle, are both comparability and co-comparability graphs.

The undirected graph $G = (V, E)$ is a *permutation graph* if there exists a pair of permutations on the vertices such the edge $\{v, v'\} \in E$ if and only if v precedes v' (or v' precedes v) in both permutations. Permutation graphs are equivalent to the comparability graphs of partial orders with dimension two. A graph is both a comparability graph and a co-comparability graph if and only if it is a permutation graph [21]. Permutation graphs, comparability graphs and co-comparability graphs are all subclasses of perfect graphs [9].

An instance of the two processor scheduling problem is given by a precedence graph $\vec{G} = (V, \vec{E})$. Eachvertex represents a task whose execution requires unit time on either of two identical processors. If there is a directed edge from task t to task t', then task t must be completed before task t' can be started. A schedule is a mapping from tasks to integer timesteps such that at most two tasks are mapped to each timestep and for all tasks t and t' if t must precede t' $(t \prec t')$ then t is mapped to an earlier timestep than t'. The length of a schedule is the number of timesteps used. An optimal schedule is one of shortest length.

The maximum matching problem on co-comparability graphs and the two processor scheduling problem are closely related. If G is a co-comparability graph and $\vec{\bar{G}}$ is a transitive orientation of G's complement, then the pairs of tasks mapped to the same timestep in an optimal two processor schedule of $\vec{\bar{G}}$ correspond to a maximum matching in G. Furthermore, there is a sequential algorithm for converting any maximum matching for G into an optimal two processor schedule for $\vec{\bar{G}}$ [5]. In [24] it was conjectured that this process is inherently sequential, but with our two processor scheduling algorithm it can be solved quickly in parallel.

Theorem 1: Two processor *scheduling is in $\mathcal{NC}$.*

Proof: We outline an $O(\log^2 n)$ time algorithm in section 3. Further details can be found in [12]. □

Theorem 2: There is an NC algorithm which detects if an undirected graph is *transitively orientable, and if so finds a transitive orientation.*

Proof: We present such an algorithm in sect ion 4. See also [16]. □

Corollary 2.1: *There is an NC algorithm which detects whether or not a graph is a permutation graph.*

Proof: Graph G is a permutation graph if and only if both G and $\bar{G}$ are comparability graphs [21]. Therefore, by running our transitive orientation algorithm on both G and $\bar{G}$, we can determine if G is a permutation graph. □

Corollary *2.2: There is an $\mathcal{NC}$ algorithm which finds a maximum node-weighted clique in comparability graphs.*

Proof: Given a comparability graph G, we find a transitive orientation, $\vec{G}$. Examine any k-path in $\vec{G}$. A k-path is a directed path containing exactly k vertices. Because $\vec{G}$ is transitively closed, the nodes on the k-path form a k-clique in G. Similarly, every *k* clique in G is a k-path in $\vec{G}$. Thus the problem of finding a maximum node-weighted clique in G reduces to finding a maximum weight path in $\vec{G}$. Since $\vec{G}$ is a DAG, standard parallel techniques (*i.e.,* max-plus closure) can be used to find a heaviest path in G. $\square$

Corollary 2.3: There *is* an NC algorithm *which finds* a minimal node-coloring of comparability graphs.

Proof: Given a comparability graph G, we find a transitive orientation, $\vec{G}$. We say that a vertex v is on level i in $\vec{G}$ if the longest (directed) path from v to a sink contains exactly i vertices. Clearly any pair of nodes on the same level are not adjacent in G, so they can be assigned the same color. Every node on level $i > 1$ is a predecessor of at least one node on level $i - 1$. Therefore, if $\vec{G}$ has k levels then $\vec{G}$ has a path of length k and G has a k-clique. Since no coloring can use fewer colors than the size of the largest clique, using a distinct color for every level yields an optimal coloring. $\square$

Theorem 3: Finding maximum matchings on co-comparability graphs *is in NC.*

Proof: One such algorithm is given in section 5. $\square$

This theorem is extended to nearly co-comparability graphs in section 5.

Corollary 3.1: Maximum matchings on permutation graphs *and partial orders of* dimension 2 can *be* constructed in NC.

Proof: As stated above, these graphs are co-comparability graphs. $\square$

Corollary 3.2: Maximum matchings *on* interval graphs can *be* found in NC.

Proof: Interval graphs are a subclass of co-comparability graphs [8]. $\square$

3 Two Processor Scheduling

In this section, we consider the scheduling problem for task systems with arbitrary precedence constraints, unit execution time per task, and two identical processors. Our scheduling algorithm for this problem is built around a routine that, for any precedence graph, computes the length of the graph's optimal schedule(s). This

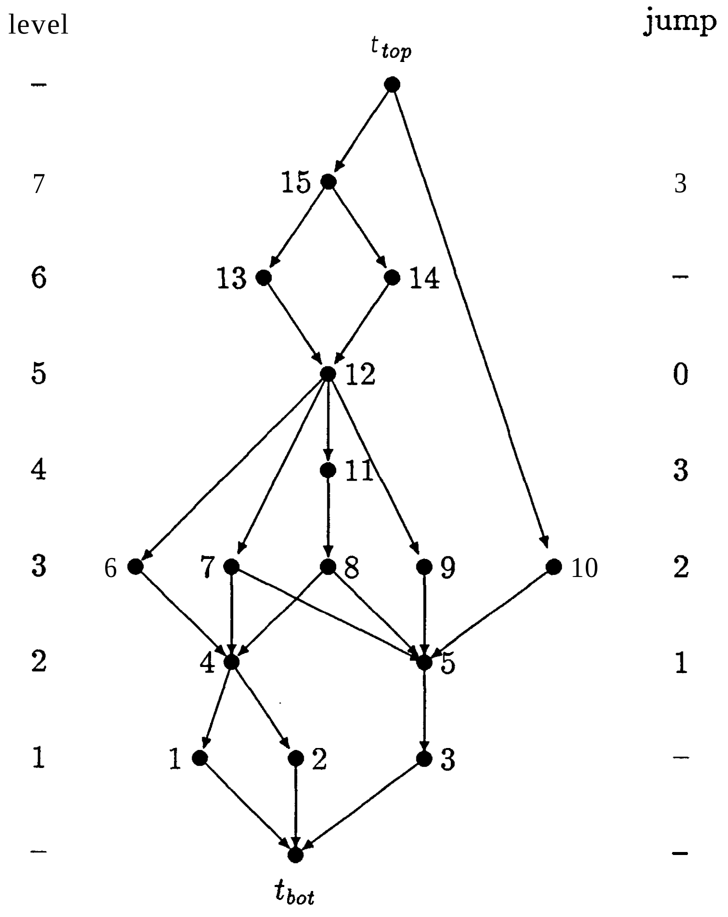

Figure 1: This is a precedence graph containing fifteen tasks (transitive arcs have been omitted). The special tasks t_{top} and t_{bot} are added when computing the length of optimal schedules for G. The levels of the original graph are on the left and the jump sequence is on the right.

length routine is applied repeatedly in order to actually find an optimal schedule for the input graph.

Let G = $(V, \prec)$ be the precedence graph we are interested in. If $t \prec t'$ then t is a predecessor of t' and t' is a successor of t. For any pair of tasks, $t, t' \in V$, define $V_{t'}^{t}$ to be the set of tasks which are both successors of t and predecessors of t', and let $G_{t'}^{t}$ be the subgraph of G induced by $V_{t'}^{t}$. The *schedule distance* between tasks t and t', $SD(t, t')$, is defined to be the length of any optimal schedule for $G_{t'}^{t}$. If $t \not\prec t'$ then $SD(t, t') = 0$.

Lemma 3.1: *Let $t, t' \in V$, and let S be a set of tasks such that for all $\hat{t} \in S$:*

i. $t \prec \hat{t} \prec t'$;

ii. $\mathrm{SD}(t, \hat{t}) \geq k$; and

```
$d_0(\star,\star) := 0;$
for $i := 1$ to $\lceil \log n \rceil$ do
    for all $t, t'$ with $t \prec t'$ do in parallel
        for all $0 \leq k, l < n-1$ do in parallel
            $S_{t,t',k,l} := \{s : t \prec s \prec t',$
                              $d_{i-1}(t,s) \geq k,$
                              $d_{i-1}(s,t') \geq l\};$
        $d_i(t,t') := \max_{S_{t,t',k,l} \neq \emptyset} \{d_{i-1}(t,t'),$
                              $k + l + \lceil |S_{t,t',k,l}|/2 \rceil\};$
$SD(\star,\star) := d_{\lceil \log n \rceil}(\star,\star)$
```

Figure 2: The Distance Algorithm.

iii. $\mathrm{SD}(\hat{t}, t') \geq 1$.

Then $\mathrm{SD}(t, t') \geq k + l + \lceil |S|/2 \rceil$.

Proof: Count the number of timesteps required to schedule those tasks between t and t'. There must be at least k timesteps before the first task in S is scheduled. It takes at least $|S|/2$ timesteps to complete the tasks in S. After the last task in S has been completed, at least l additional timesteps are required. Therefore $SD(t, t') \geq k + l + |S|/2$. □

The distance algorithm (see Figure 2) uses a doubling method like transitive closure to compute the schedule distances between all pairs of tasks in a precedence graph $G = (V, \prec)$. It initially guesses that the scheduling dist ance between each pair of tasks is at least zero. By repeatedly applying Lemma 3.1 to each pair of tasks in parallel the algorithm refines its guesses. Below we prove that after log $|V|$ iterations, the algorithm's guess for each pair of tasks has converged to the schedule distance. The distance algorithm has a straightforward implementation on an n^5 processor PRAM taking $O(\log^2 n)$ time.

Lemma 3.2: *The distance algorithm* always *computes the schedule distance between every pair of tasks.*

Proof: Lemma 3.1 guarantees that the distances computed by the algorithm are never greater than the the schedule distances.

In [2] it is shown how to construct sets of tasks $\chi_0, \chi_1, \ldots, \chi_k$ for any precedence graph such that:

- Those tasks in any χ_i are predecessors of all tasks in χ_{i-1}, and
- The length of optimal schedules for G is $\sum_i \lceil |\chi_i|/2 \rceil$ (See Figure 3).

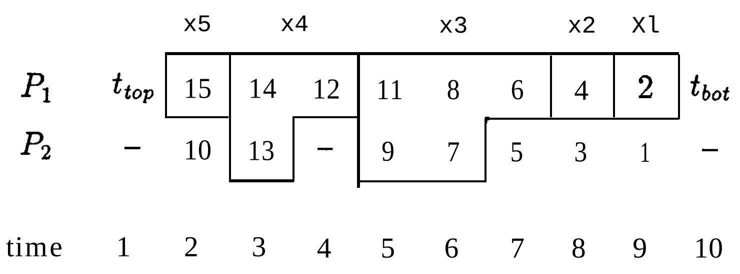

Figure 3: This is an LMJ schedule for the graph in Figure 1; each χ_i is boxed.

Our algorithm does not compute the χ_i's directly, we simply use their existence to prove that the distances the algorithm does compute converge to the scheduling distance.

Examine how the distance algorithm determines the schedule distance between an arbitrary pair of tasks, t and t'. Let $\chi_1, \chi_2, \ldots, \chi_h$ be a set of χ_i's for $G^t_{t'}$, $\chi_{h+1} = \{t\}$, and $\chi_0 = \{t'\}$. After the first iteration of the outer loop, the distance computed between any task in χ_i and one in χ_{i-2} is at least $\lceil|\chi_{i-1}|/2\rceil$. After the second iteration, the distance computed between any task in χ_i and any task in χ_{i-4} is at least $\lceil\chi_{i-1}/2\rceil + \lceil\chi_{i-2}/2\rceil + \lceil\chi_{i-3}/2\rceil$. This is an easy consequence of Lemma 3.1 with $S = \chi_{i-2}$, $k = \lceil\chi_{i-1}/2\rceil$, and $l = \lceil\chi_{i-3}/2\rceil$. In each iteration we double the number of χ_i's accounted for. After log h iterations, the computed distance between t and t' is at least the optimal schedule length for $G^t_{t'}$, and thus at least $SD(t, t')$.

Since G contains n tasks, each $G^t_{t'}$ has at most $n - 2$ χ_i's. Therefore, after $\lceil\log n\rceil$ iterations the algorithm has converged to the schedule distances for each pair of tasks. $\square$

The distance algorithm can be used to compute the length of optimal schedules for a graph. Augment the graph with two dummy tasks, t_{top} and t_{bot}, which are a predecessor and successor (respectively) of all other tasks in G. Now $SD(t_{\text{top}}, t_{\text{bot}})$ is the length of G's optimal schedules, and can be found using the distance algorithm.

The method for converting the distance algorithm into one which finds an optimal schedule involves several constructions. For the sake of brevity this paper contains only an outline of our method. Interested readers may consult [12] for a more detailed present at ion.

The search for an optimal schedule can be restricted to the class of *Lexicographically Maximal Jump* (LMJ) schedules. Each task t in the precedence graph is assigned a *level* equal to the number of tasks in the longest path from t to a sink. *A level schedule* gives preference to tasks on higher levels. More precisely, suppose levels $L, \ldots, l+1$ have already been scheduled and there are k unscheduled tasks remaining on level l. If k is even a level schedules puts the k tasks in pairs, and there is no jump from level l. If k is odd, a level schedule pairs $k-1$ of the tasks with each other and the remaining task is paired with a task from a lower level $l' < l$. In this case, level l *jumps* to level l'. We assume that there are an unlimited number of dummy tasks on level 0 which can be paired with any other tasks. The jump *sequence* of a level schedule is the sequence of levels jumped *to,* listed in the order in which the jumps occur (see Figure 1). The *Lexicographically Maximum Jump (LMJ)* sequence is the jump sequence (resulting from some level schedule) that is lexicographically greater than any other jump sequence resulting from a level schedule. An *LMJ schedule* is a level schedule whose jump sequence is the LMJ sequence. Note that our definition of LMJ is similar to the definitions of highest level first in [6] and [24]. The following theorem establishes the importance of LMJ schedules.

Theorem 4: [6] *Every LMJ schedule* is optimal. $\square$

Our two processor algorithm uses the distance algorithm to find the LMJ sequence and which jump (if any) a pair of tasks can be used for. In general, there will be many possible pairs for each jump. A path doubling computation finds a consistent set of task pairs for the jumps. The remaining tasks are paired up within levels. Since there are never precedence constraints between two tasks on the same level, this pairing can be done arbitrarily. An LMJ schedule is obtained by sorting the resulting set of task pairs (both for jumps and within levels). We refer the reader to [12] for a complete description of the technically more involved parts of this construction.

4 Transitive Orientation

The transitive orientation problem is nontrivial because some edges cannot be oriented independently. If the edges $\{a, b\}$ and $\{b, c\}$ are in the graph to be oriented, but the edge $\{a, c\}$ is not, then the edges $\{a, b\}$, $\{b, c\}$ cannot be oriented independently. If we choose the arc (a, b) then we are forced to include the arc (b, c) in the transitive orientation (see Figure 4). The binary relation Γ reflects this simple kind of forcing. [21]. Given $G = (V, E)$, we say that $(a, b)\Gamma(a, c)$ and $(b, a)\Gamma(c, a)$ whenever $\{a, b\} \in E$, $\{a, c\} \in E$ and $\{b, c\} \notin E$.

The reflexive, transitive closure of Γ, Γ^*, is an equivalence relation on the possible orient at ions of edges in E. For obvious reasons, we call these equivalence classes *implication classes.* If $\vec{A}$ is a set of arcs (e.g. an implication class) then A denotes the set of undirected edges $\{\{a,b\} : (a, b) \in \vec{A} \vee (b,a) \in \vec{A}\}$, and $\vec{A}^{-1}$ is the set of arcs $\{(b,a) : (a, b) \in \vec{A}\}$. A set of arcs $\vec{A}$ is *consistent* if $\vec{A} \cap \vec{A}^{-1} = \emptyset$, and is *inconsistent* when $\vec{A} \cap \vec{A}^{-1} \neq \emptyset$.

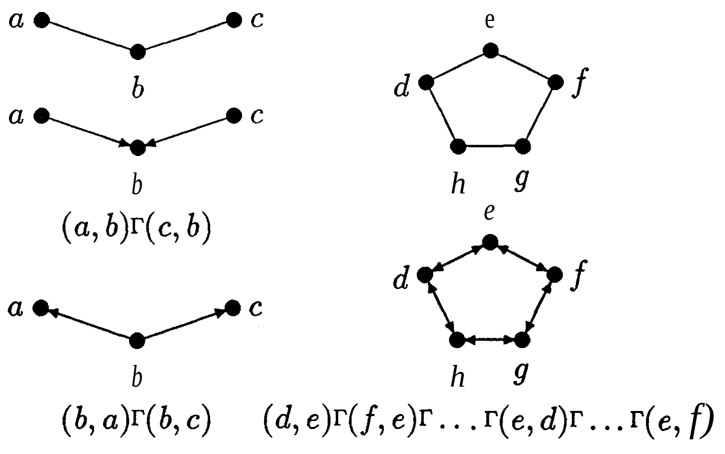
a
c
b
a
c
b
$(a,b)\Gamma(c,b)$
a
c
b
$(b,a)\Gamma(b,c)$
e
d
f
h
g
e
d
f
h
g
$(d,e)\Gamma(f,e)\Gamma\ldots\Gamma(e,d)\Gamma\ldots\Gamma(e,f)$

Figure 4: Graphs and Implication Classes

Implication classes have been studied by M. C. Golumbic and many of the lemmas in this section have originally been shown in [9] or [10].

Lemma 4.1: *If $\vec{A} \neq \vec{B}$ are implication classes of G then either $\vec{A} = \vec{B}^{-1}$ or $A \cap B = \emptyset$.*

Proof: Assume that $\{a, b\} \in A \cap B$. Without loss of generality, let $(a, b) \in \vec{A}$. If $(a, b) \in \vec{B}$ then $\vec{B} = \vec{A}$ since implication classes are equivalence classes. Therefore $(b, a) \in \vec{B}$, and $(b, a) \notin \vec{A}$. By definition, if $(a, b)\Gamma(a', b')$ then $(b,a)\Gamma(b', a')$. Thus some $(c,d)\Gamma^*(a, b)$ if and only if $(d,c)\Gamma^*(b,a)$, so $\vec{A} = \vec{B}^{-1}$. □

Given an undirected graph $G_1 = (V, E)$ pick any implication class $\vec{B_1}$, delete B_1 from G_1, forming $G_2 = (V, E - B_1)$. Next form G_3 by removing the underlying set B_2 of some implication class $\vec{B_2}$ of G_2. Continue the process until removing B_k from G_k results in a graph with no edges. The sequence of implication classes removed, $\vec{B_1}, \vec{B_2}, \ldots, \vec{B_k}$, is called a *$\Gamma$-decomposition* of G. The following theorem points out the usefulness of Γ-decompositions.

Theorem 5: *(TRO Theorem [9]) Let $\vec{B_1}, \vec{B_2}, \ldots, \vec{B_k}$ be a Γ-decomposition of an undirected graph G. The following statements are equivalent:*

i. *G is a comparability graph.*

ii. *Every implication class of G is consistent.*

iii. Each *$\vec{B_i}$ in the Γ-decomposition is consistent.*

Furthermore, when these conditions hold, $\vec{B_1} \cup \vec{B_2} \cup \ldots \cup \vec{B_k}$ is a transitive *orientation of G.*

Proof: The proof of this theorem requires several technical lemmas, and thus is beyond the scope of this paper. The interested reader is referred to [9,10]. $\square$

Let $\vec{A}$ be any implication class of the graph G. Then we call the underlying set of edges A its color class. The TRO theorem suggests a sequential algorithm for finding transitive orientations of comparability graphs. One can take any edge, orient it arbitrarily, find the associated implication class, add the implication class to the transitive orientation and remove its color class from the comparability graph. Repeating this procedure yields a Γ-decomposition of the comparability graph and therefore a transitive orientation. This is essentially the algorithm in [21].

If we are dealing with a comparability graph it is sufficient to consider color classes instead of implication classes, since every color class A represents and implication class $\vec{A}$ and its inverse $\vec{A}^{-1}$. When talking about color classes we always assume that the corresponding implication classes are consistent.

In order to parallelize the sequential algorithm above it is neccessary to understand how color classes change during a I'-decomposition. We will see below that the changes are very simple: color classes are either merged with other color classes or remain unchanged.

Lemma 4.2: *Let B be an color class of $G = (V, E)$. Every implication class of $G' = (V, E - B)$ is the union of color classes of G.*

Proof: The Γ relation for G', restricted to $E - B$, contains the corresponding restriction of the Γ relation for G. $\square$

The three edges of a triangle in the undirected graph G form a tricolored triangle if they belong to three distinct color classes. We say that two color classes A and B are *triangle related,* written $A \triangle B$, if there is a tricolored triangle in G with one edge in A and another edge in B.

Lemma 4.3: *Let A and B be two distinct color classes in $G = (V, E)$. A is* not *an implication class of $G' = (V, E - B)$ iff $A \triangle B$.*

Proof: The proof is a simple consequence of the definition of the Γ relation. It will be omitted here. $\square$

An immediate implication of Lemma 4.3 is

Lemma *4.4: Let the color classes $B_1, \ldots, B_k$ of $G = (V, E)$ be an independent set under the $\triangle$ relation. Then in $G' = (V, E - B_1)$, the collection $\{B_2, \ldots, B_k\}$ is an independent set under $\triangle$.*

Corollary 4.4.1: *If color classes $B_1, \ldots, B_k$ of G form an independent set under the $\triangle$ relation, then they are the first k color classes for a IT-decomposition of G.*

Proof: Follows from the definition of independent set. □

Lemma 4.5: *Let $B_1, \ldots, B_k$ be* a maximal independent *set* under *the* $\triangle$ relation *for some* graph $G_1 = (V, E)$. *Every color class of $G_{k+1} = (V, E - B_1 - B_2 - \ldots - B_k)$* is *the* union *of at least two color* classes *of* G_1.

Corollary 4.5.1: *The* number of *color* classes *for* G_{k+1} is *at most* half *the* number *of color classes for G.*

Proof: Since the B_i form a maximal independent set under $\triangle$ every color class of G which is not one of the B_i must be adjacent to one of the B_i. Because of Lemma 4.3 it will be merged with some other color class. □

The input to our algorithm is an undirected graph G = *(V, E)*. The output is either $\vec{G}$, a transitive orientation of G, or an indication that G has no transitive orientation. With G_1 initialized to be G, and $\vec{G}$ initially equal to $(V, \emptyset)$, if no inconsistent implication class is found in the first iteration, the algorithm proceeds in iterations as long as the set of color classes is non-empty.

Each iteration consists of the following four steps:

1. Determine the color classes of G_i. This can be done using standard parallel techniques such as solving 2-SAT formulae or finding connected components [22].
2. Determine the $\triangle$ relation on color classes.
3. Use a maximal independent set subroutine [15,17] to obtain a maximal independent set M of color classes.
4. In parallel, for each B_j in M, delete B_j from G_i, and add $\vec{B_j}$ or $\vec{B_j}^{-1}$ to $\vec{G}$.

Step 3 is the most expensive of these steps, requiring $O(\log^2 n)$ time and n^4 processors. The logn iterations can therefore be done in $O(\log^3 n)$ time on n^4 processors.

5 Maximum Matching

The two processor scheduling and transitive orientation algorithms can be used to find maximum matchings on co-comparability graphs. To find a maximum matching on the co-comparability graph G = *(V, E),* first create the comparability graph G, the complement of G. Applying the transitive orientation routine converts $\bar{G}$ into a precedence graph. An optimal two processor schedule can be found for the precedence graph using our scheduling algorithm. We will see below that the pairs of tasks scheduled together form a maximum matching of G.

Let S be any optimal two processor schedule for $\vec{\bar{G}}$. A *task-pair* of S is a pair of tasks mapped to the same timestep by S. Since there are no precedence relationships between tasks in a task-pair, the set of task-pairs of S form a matching in G. Because S is an optimal schedule, no schedule has more task-pairs.

A task is *available* at some point in a schedule if it can be executed without violating the precedence constraints.

Lemma *5.1: If a co-comparability graph G has a perfect matching then $\vec{\vec{G}}$ has a schedule where every task is in a task-pair.*

Proof: We say a pair of tasks is mated if the pair is in the perfect matching. Construct the schedule (and modify the "mated" relationship) iteratively as follows:

> If two mated tasks are both available, schedule one such mated pair. Otherwise find two mated pairs, (t, t') and (s, s'), such that t and s are available and there is no precedence relationship between t' and s'. Schedule t with s and mate t' with s'.

Note that there are never precedence constraints between a pair of mated tasks. This method clearly takes two tasks each timestep and does not violate the precedence constraints. What we must show is that it always constructs a schedule for $\vec{\vec{G}}$.

Assume to the contrary that at some point it does not find a pair of tasks to schedule. Let U be the set of available tasks and U' be the set of tasks which are mated to tasks in U. Since the method fails, $U \cap U' = \emptyset$ and there is a precedence relationship between every pair of tasks in U' (i.e. U' is totally ordered). Let t' be the task in U' which precedes all other tasks in U'. Since $t' \notin U$, there must be some $t \in U$ such that $t \prec t'$. However, by the transitivity of precedence, t also precedes its mate – contradiction. □

Lemma *5.2: Let $\vec{\vec{G}} = (V, \prec)$ be a precedence graph and S a two processor schedule for $\vec{\vec{G}}' = (V - \{t\}, \prec)$. A single timestep containing t can be inserted into S yielding a schedule for $\vec{\vec{G}}$.*

Proof: Let t' be the last predecessor of t in S. Insert task t immediately after the timestep containing t'. Obviously there are no precedence conflicts between t and its predecessors. Since S is valid schedule, there are no precedence conflicts between tasks in $V - \{t\}$. Therefore any precedence conflict which is violated is of the form $t \prec \hat{t}$. By transitivity t' also precedes $\hat{t}$, so $\hat{t}$ comes strictly after t' in S. Since t is inserted immediately after t', task t appears before $\hat{t}$ in the modified schedule. □

Let M be the tasks in a maximum matching on G. The above Lemmas suggest a way to obtain a schedule, S, for $\vec{\vec{G}} = (V, \prec)$ where the paired tasks of S are precisely the tasks in M. Start by finding an optimal schedule, S' for the subgraph of $\vec{\vec{G}}$ induced by M and add the tasks in V-M one at a time. One **NC** implementation of this algorithm involves bucket sorting the tasks in $V - M$ based on which task-pair of S' they follow. By topologically sorting the tasks within each bucket we can quickly determine where each task should be inserted.

Theorem 6: *The* task-pairs *of any* optimal schedule *for* $\vec{\vec{G}}$ *form* a maximum *matching on G.*

Proof: Let M be the tasks in some maximum matching of G. Let S be an optimal schedule for the subgraph of $\vec{\vec{G}}$ induced by M. By Lemma 5.1, the task-pairs of S form a maximum matching on G. By Lemma 5.2 we can insert the other tasks of $\vec{\vec{G}}$ one at a time without disturbing the task-pairs. Therefore, the task-pairs of the resulting schedule for $\vec{\vec{G}}$ form a maximum matching on G. Since every optimal schedule has the same number of task-pairs and the task-pairs of every schedule form a matching, the task-pairs of any optimal schedule for $\vec{\vec{G}}$ forms a maximum mat ching on G. □

If $\bar{G}$ is not transitively orientable it may still be possible to find a maximum matching in G = (V, E). Assume we are given a set U, consisting of O(log n) edges, such that $\bar{G} \cup U$ is transitively orientable. The following method finds a maximum matching in G.

For each S' $\subseteq U$ such that S' is a matching find (in parallel) a maximum matching in G' = $(V - \{v : (v,v') \in S'\}, E - U)$. A maximum matching for G occurs whenever the cardinality of the maximum matching for G' plus $|S'|$ is maximal.

A graph G is a k-nearly comparability graph when:

— G has at most k log n inconsistent implication classes and

— each inconsistent implication class of G is split into consistent implication classes by the addition of at most k edges.

A k-nearly co-comparability graph is the complement of a k-nearly comparability graph.

Let G be a k-nearly co-comparability graph (for some constant k). The following is an outline of an NC algorithm for finding a maximum matching in G. In parallel examine each set, T, of k edges not in $\bar{G}$. Determine which inconsistent implication classes are split when T is added to $\bar{G}$. For each inconsistent implication class $\vec{A}$, pick any set of k edges which splits $\vec{A}$ into consistent implication classes. At most $k^2 logn$ edges are picked, so the above method can be used to find a maximum matching for G.

6 Conclusions

Although the algebraic approach was used to obtain the first parallel matching algorithms [14,18], these are randomized algorithms. It is interesting that we can obtain deterministic matching algorithms for wide classes of graphs using a purely combinatorial approach. Perhaps the combinatorial approach will yield deterministic algorithms for matching on other classes of graphs as well.

It was surprising how much more difficult computing the actual schedule was than simply computing its length. In higher complexity classes such as $\mathcal{P}$ and $\mathcal{NP}$ it is often easy to go from the decision problem to computing an actual solution, because of self reducibility. However this does not necessarily seem to be the case for parallel complexity classes. To support this observation we note that the random NC algorithm for finding the cardinality of a maximum matching is much simpler than the random NC algorithm for determining an actual maximum matching [13].

There are several open problems related to scheduling. We are attempting to extend our two processor result to the case when the tasks have nonuniform start times and/or deadlines. When the precedence constraints are restricted to in-trees or out-trees there are parallel algorithms for generating schedules on an arbitrary number of processor [3,11]. It is an open problem whether interval-ordered tasks [19] can be scheduled in parallel.

One variant of the two processor problem that we know to be NP-complete (by reduction from the clique problem) allows incompatibility edges as well well as precedence constraints. When there is an incompatibility constraint between two tasks they can be executed in either order, but not concurrently. Incompatibility constraints arise naturally when two or more tasks need the same resource, such as special purpose hardware or a database file.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, New York, 1974.

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

[3] D. Dolev, E. Upfal, and M. Warmuth. Scheduling trees in parallel. In *Bertolazzi, P, Luccio, F. (eds.): VLSI: Algorithms and Architectures. Proceedings of the International Workshop on Parallel Computating and VLSI,* pages L-30, North-Holland, 1985.

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

[5] M. Fujii, T. Kasami, and K. Ninamiya. Optimal sequencing of two equivalent processors. *SIAM J. Appl. Math.,* 17(4):784–789, *1969.*

[6] H. Gabow. An almost-linear algorithm for two-processor scheduling. *JACM,* 29(3):766–780, *1982.*

[7] H. Gabow and R. Tarjan. A linear time algorithm for special case of disjoint set union. In *Proceedings of the 15th Ann. ACM Symp. on Theory of Computing (Boston, Mass.),* pages 246-251, 1983.

[8] P. Gilmore and A. Hoffman. A characterization of comparability graphs and of interval graphs. *Canad. J. Math,* 16, 1964.

[9] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[10] M. Golumbic. Comparability graphs and a new matroid. *J. Combinatorial Theory (B),* 22(1):68–90, 1977.

[11] D. Helmbold and E. Mayr. Fast scheduling algorithms on parallel computers. *Advances in Computing Research,* 1986. to appear.

[12] D. Helmbold and E. Mayr. Two processor scheduling is in $\mathcal{NC}$. In *Proc. 1986 Aegean Workshop on computing: VLSI Algorithms and Architectures,* July 1986.

[13] R. Karp, E. Upfal, and A. Wigderson. Are search and decision problems comput at ionally equivalent? In *Proceedings of the 17th Ann. ACM Symp. on Theory of Computing (Providence, RI),* pages 465-475, 1985.

[14] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. In *Proceedings of the 17th Ann. ACM Symp. on Theory of Computing (Providence, RI),* pages 22-32, 1985.

[15] R. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. ACM,* 32(4):762–773, 1985.

[16] D. Kozen, U. Vazirani, and V. Vazirani. NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching. In *5th Conf. Found. of Software Tech. and Theor. Comp. Sci. (New Dehli),* 1985.

[17] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Ann. ACM Symp. on Theory of Computing (Providence, RI),* pages 1-10, 1985.

[18] K. Mulmuley, U. Vazirani, and V. Vazirani. Parallel algorithms for rank and mat ching. private communication.

[19] C. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Computing,* 8(3), 1979.

[20] N. Pippenger. On simultaneous resource bounds. In Proceedings *of the 20th IEEE Symp. on Foundations of Computer Science,* pages 307-311, 1979.

[21] A. Pnueli, A. Lempel, and S. Even. Transitive orientation of graphs and identification of permutation graphs. *Can. J. Math.*, 23(1):160–175, 1971.

[22] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–63, 1982.

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

[24] U. Vazirani and V. Vazirani. The two-processor scheduling problem is in $\mathcal{RNC}$. In *Proceedings of the 17th Ann. ACM Symp. on Theory of Computing (Providence, RI)*, pages 11-21, 1985.