Parallel Algorithms For Maximum Matching And Other Problems On Interval Graphs

Abha Moitra*
Richard Johnson*

TR 88-927
July 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Research supported in part by the National Science Foundation under grant DCI-86-02256.

Parallel Algorithms For Maximum Matching And Other Problems On Interval Graphs

Abha Moitra[1] Richard Johnson [1]
Dept. of Computer Science, Cornell University, Ithaca NY 14853

Abstract: In this paper, we consider parallel algorithms on *interval graphs.* An interval graph is a graph having a one-to-one correspondence with a sequence of intervals on the real line, such that each vertex maps to an interval in the sequence and an edge exists between two vertices if and only if the corresponding intervals overlap. Throughout the paper we use the CREW PRAM model. Our main result is an $O(\log^2 n)$ time, $O(n^6/\log n)$ processor algorithm for maximum matching on interval graphs. We give PT-optimal algorithms for maximum weighted clique, maximum independent set, minimum clique cover, and minimum dominating set for representations of interval graphs; and Hamiltonian circuit for representations of proper interval graphs. We also give an improved algorithm for minimum bandwidth on representations of proper interval graphs. In addition, we present $O(\log n)$ time, $O(n^2/\log n)$ processor algorithms for depth-first search on representations of interval graphs and maximum matching on representations of proper interval graphs.

1 Introduction

In this paper, we consider parallel algorithms on *interval graphs.* An interval graph is a graph having a one-to-one correspondence with a sequence of intervals on the real line, such that each vertex maps to an interval in the sequence and an edge exists between two vertices if and only if the corresponding intervals overlap. We denote the interval set as $I = \{i_1, i_2, \ldots, i_n\}$ where i_k denotes the real line interval $[a_k, b_k]$. An interval graph is *proper* if no interval is properly contained within some other interval. In this paper, we will refer to interval i, interval $[a_i, b_i]$, and vertex i (corresponding to interval i) interchangeably. Given a graph, its interval representation (if one exists) may be obtained on the CRCW PRAM model with the $O(\log^2 n)$ time and $O(n+m)$ processor algorithm of Klein [11] or with the $O(\log n)$ time and $O(n^3)$ processor algorithm of Novick [14]. There are many applications of interval graphs, spanning the study of problems in genetics to archeology; for more details see Golumbic [8] where he says "interval graphs are among the most useful mathematical structures for modeling real world problems." In many applications, a representation of an interval graph is part of the problem statement, and hence it is natural to study the complexity of parallel algorithms on representations of interval graphs.

Our main result is an $O(\log^2 n)$ time, $O(n^6/\log n)$ processor algorithm for maximum matching on interval graphs. The only previous parallel algorithm for this is a byproduct of a parallel algorithm for two-processor scheduling by Helmbold and Mayr [10]. They showed that two-processor scheduling is in $\mathcal{NC}$ and gave an $O(\log^2 n)$ time, $O(n^{10})$ processor algorithm. Since any optimal two-processor schedule on a precedence graph corresponds to a maximum matching on its complement, and the complement of an interval graph is

[1] Research supported in part by National Science Foundation grant DCI-86-02256.

a precedence graph, it followed that maximum matching on interval graphs could be done in $O(\log^2 n)$ time using $O(n^{10})$ processors. By foc using on the special structure of interval graphs, we will show how this result can be improved considerably.

We also improve several parallel algorithms from Bertossi and Bonuccelli [2] by giving PT-optimal algorithms for maximum weighted clique, maximum independent set, minimum clique cover, and minimum dominating set for representations of interval graphs; and Hamiltonian circuit for representations of proper interval graphs. We also give an improved algorithm for minimum bandwidth for representations of proper interval graphs and present $O(\log n)$ time, $O(n^2/\log n)$ processor algorithms for depth-first search on representations of interval graphs and maximum matching on representations of proper interval graphs.

2 Correspondence Between Maximum Matching and Two-Processor Scheduling

In this section we show how maximum matching on an interval graph corresponds to an optimal two-processor schedule on a suitable graph. Consider a directed acyclic graph (dag) $G = (V, E)$. A (two-processor) schedule on G can be defined as an assignment of the elements of V to time units $1, 2, \ldots, w$ so that

- each element of V is assigned exactly one time unit,
- at most two elements of V are assigned the same time unit, and
- if there is a path from x to y in G, then x is assigned a lower time unit than y.

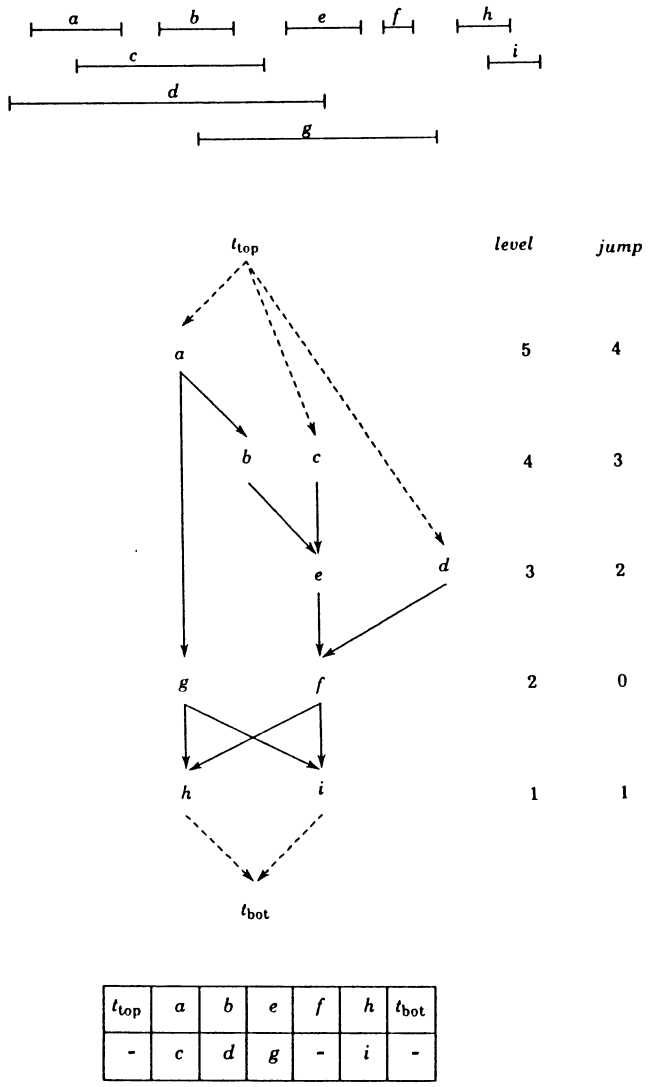
That is, in G the vertices represent unit time tasks and the edges specify the precedence constraints among the tasks. The length of the schedule is w and an optimal schedule minimizes w. In a schedule, if two tasks are assigned the same time unit those tasks are said to be paired or matched. The connection between optimal schedule and maximum matching is as follows.

Theorem 1: (Fujii, Kasami, and Ninomiya [6]) Let $G = (V, E)$ be a dag and let $\overline{G^*}$ be the complement of its transitive closure. Then the paired tasks in an optimal schedule for G form a maximum matching in $\overline{G^*}$.

Since interval graphs are properly contained in $\overline{G^*}$ [12], it follows that a maximum matching for an interval graph can be obtained from an optimal schedule on a suitable dag. Figure 1 gives an example of an interval graph, its corresponding dag (solid edges), and its optimal schedule where maximum matching for the interval graph corresponds to an optimal schedule for the dag. In general, given a representation for an interval graph $G = (\{[a_i, b_i] \mid i = 1, \ldots, n\}, E)$, we can construct a corresponding dag as follows:

$$G' = (\{1, 2, \ldots, n\}, E') \text{ where } E' = \{\langle i, j\rangle \mid 1 \leq i, j \leq n, i \neq j, \text{ and } b_i < a_j\} \quad (1)$$

Since we will consider only dags related to interval graphs, we define $\text{DAG}_\text{I} = \{G' \mid \exists$ interval graph G such that G' is constructed from G as in (1) $\}$.



t_{top}	a	b	e	f	h	t_{bot}
-	c	d	g	-	i	-

Figure 1: An interval graph, its corresponding dag, LMJ schedule, and jump sequence

3 Two-Processor Scheduling

In this section we introduce relevant terminology and results for two-processor scheduling. We will also present a parallel algorithm for obtaining the length of an optimal schedule which will be used later to obtain the parallel algorithm for maximum matching on interval graphs.

3.1 Terminology

Throughout this section we will assume for simplicity that $G = (V, E)$ is a dag where all the transitive edges have been dropped. We write $x \prec y$ if x is a predecessor of y in G. In a schedule S for G, if only one task is assigned at time unit t, then schedule S is said to have an empty slot at time t.

The level of a task x in G, $level(x)$, is the number of tasks on the longest path from x to a sink, and $level_l$

is the set of all tasks at level l. A *level schedule* is one that maps tasks of higher levels first. That is, if the highest level is L and the tasks from levels $L, L-1, ..., l+1$ have already been mapped and k unmapped tasks on level l remain, then the schedule will map these k tasks to the next $\lceil k/2 \rceil$ time units. If k is odd then the last time unit containing a task from level l may have either an empty slot or a task from a lower level. In a level schedule, level l is said to jump to level $l', l \geq l'$, written as $Jump(l) = l'$, if the last time unit containing a task from level l also contains a task from level l'. If the last time unit containing a task from level l has an empty slot, then $Jump(l) = 0$. The jump sequence of a level schedule is $\langle Jump(L), Jump(L-1), .., Jump(1) \rangle$ where L is the highest level in G. An *LMJ schedule* is a level schedule with lexicographically largest jump sequence. Figure 1 gives an example of a level schedule and its jump sequence. The importance of an LMJ schedule is:

Theorem 2: (Gabow [7]) Every LMJ schedule is optimal.

3.2 Parallel Algorithm for Length of Optimal Schedule

We now describe how the length of an optimal schedule can be obtained. The distance algorithm of Helmbold and Mayr [10] given below computes the length of the optimal schedule by first adding two tasks, t_{top}, t_{bot} to G where t_{top} (t_{bot}) is the predecessor (successor) of all tasks. The *schedule distance* between tasks t, t', $\mathrm{D}(t, t')$, is the number of time units required to schedule all tasks that are both successors of t and predecessors of t'. If $t \not\prec t'$ then $\mathrm{D}(t, t') = 0$. $\mathrm{D}(t_{top}, t_{bot})$ is then the length of an optimal schedule for G, denoted opt(G).

Theorem 3: (Helmbold and Mayr [10]) Algorithm *distance* computes the length of an optimal schedule for a dag G in $O(\log^2 n)$ time using $O(n^5/\log n)$ processors on the CREW PRAM model.

Algorithm *distance*

for all t, t', $d_0(t, t') := 0$;
for $r := 1$ **to** $\lceil \log n \rceil$ **do**
  for all t, t' with $t \prec t'$ **do in parallel**
    for all $0 \leq j, k < n-1$ **do in parallel**
      $U_{t,t',j,k} := \{\hat{t} : t \prec \hat{t} \prec t', d_{r-1}(t, \hat{t}) \geq j, d_{r-1}(\hat{t}, t') \geq k\}$;
    $d_r(t, t') := \max_{U_{t,t',j,k} \neq \emptyset} \{d_{r-1}(t, t'), j + k + \lceil |U_{t,t',j,k}|/2 \rceil\}$;
for all t, t', $\mathrm{D}(t, t') := d_{\lceil \log n \rceil}(t, t')$

(Note, in [10] the number of processors used is actually $O(n^5)$ but can be reduced by standard techniques.)

4 Sequential Maximum Matching on Interval Graphs

In this section we show that a sequential greedy algorithm generates a maximum matching on an interval graph. This fact will be used in Section 5 to develop a parallel algorithm for maximum matching on interval graphs.

Algorithm *Greedy*: /* Input is I, a representation of an interval graph */

$$\text{Let } match(i) = \begin{cases} \text{first ending interval in } I \text{ which overlaps interval } i. \\ 0, \text{ otherwise.} \end{cases}$$

$M = \emptyset$;
while $I \neq \emptyset$ **do**
  $i :=$ first ending interval in I;
  $I := I - \{i\}$;
  if $match(i) \neq 0$ **then**
    add $(i, match(i))$ to M;
    $I := I - \{match(i)\}$

Theorem 4: The *Greedy* algorithm generates a maximum matching that corresponds to an LMJ schedule.

proof: Let M be the matching produced by the *Greedy* algorithm. We first prove that M corresponds to a level schedule. From the construction of the dag corresponding to a representation of an interval graph in Section 2 and the definition of levels in Section 3.1 it follows that if level(i) > level(j) then $b_i < b_j$. So the *Greedy* algorithm considers the vertices on the highest unfinished level first. Hence M corresponds to a level schedule.

Now suppose M does not correspond to an LMJ schedule. Let $M_{seq} = \langle a_1, a_2, \ldots, a_n \rangle$ denote the order in which vertices are removed from I by *Greedy*. Define the *correspondence* of a maximum matching M' with M to be $\max_i \{a_i \mid \forall j < i, a_j$ matches the same in M and $M'\}$. Choose an LMJ schedule $\hat{M}$ with maximum correspondence with M. Let a be the first vertex in M_{seq} which matches differently from M and $\hat{M}$.

case I: a does not have a match in M but has a match in $\hat{M}$. This is impossible because if a can be matched with b, a will be matched with something in M.

case II: a is matched in M but not in $\hat{M}$. Let b be the vertex to which a is matched in M. If a and b are in the same level it would imply that $\hat{M}$ is not a level schedule - a contradiction. So, level(b) < level(a). But that implies that the jump sequence for M is greater than that of $\hat{M}$ which is again a contradiction.

case III: a is matched in M and in $\hat{M}$, but to different vertices. Let a be matched to b in M and to c in $\hat{M}$. Since $\hat{M}$ is an LMJ schedule, level(c) $\geq$ level(b). Since b ends before c it follows that the maximum number of independent intervals beginning after b is no less than the maximum number of independent intervals beginning after c and hence level(b) $\geq$ level(c). Therefore, level(b) = level(c). Since b ends before c, every vertex not yet considered that can match with b can also match with c. Hence we can construct a new LMJ schedule by interchanging b and c in $\hat{M}$ which would be closer to M, a contradiction. □

The consequence of Theorem 4 is that for any dag $\in \mathrm{DAG_I}$, we can efficiently determine the actual tasks which perform the level jumps. More specifically, if level l jumps to level l' and if S_l is the set of remaining tasks at level l when processing is to begin for level l, then the task to jump to level l' is the last ending task in S_l. (This is in contrast to the condition in Section 3 where the jump had to be chosen so as to maximize the ensuing jump sequence.) We will be able to make use of this property in the next section to obtain a more efficient parallel algorithm for the matching problem on interval graphs.

5 Maximum Matching on Interval Graphs

In this section we will show how the properties of interval graphs can be exploited to efficiently determine all the pairs of tasks involved in an optimal schedule for dags in $\mathrm{DAG_I}$.

5.1 Parallel Algorithm for Maximum Matching on Interval Graphs

Our algorithm will proceed by finding an optimal schedule for the dag corresponding to the given interval graph. It will first identify, for each level l, a set of tasks from that level that are used in actual jumps to level l. Then, all the pairs of tasks constituting a jump will be identified. This information will then be used to construct an optimal schedule. We first introduce some terminology for dags.

Definition 1: Level l is said to be a 1-level if in any LMJ schedule, level l jumps to a lower level.

Definition 2: G_l is the dag G with all tasks below level l removed and G'_l is the dag G_l with one additional task added to the lowest level and made successor to all tasks on higher levels.

Algorithm MM

Given an interval graph $G' = (V, E)$ with its representation $I = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ and dag $G \in \mathrm{DAG_I}$ with L levels corresponding to G', construct an optimal two-processor schedule as follows:

Step 1 For each level l determine if it is a 1-level. Note l is a 1-level iff $\mathrm{opt}(G_l) = \mathrm{opt}(G'_l)$.

Step 2 For each level l, determine

$$\mathrm{Cand}_l = \{x \mid x \in \mathit{level}_l, \exists \text{ an optimal schedule } S \text{ for } G_l \text{ where } x \text{ is scheduled in the last time slot in } S\ \}$$

This can be done as follows:

If l is a 1-level

then $\mathrm{Cand}_l = \{x \mid x \in \mathit{level}_l,\ \mathrm{opt}(G_l) = \mathrm{opt}(G_l - \{x\}) + 1\ \}$

else $\mathrm{Cand}_l = \{x \mid x \in \mathit{level}_l,\ \mathrm{opt}(G_l) = \mathrm{opt}(G_l - \{x\})$ and l is a 1-level in $G_l - \{x\}\}$

Step 3 Let last(Cand_l) be the 'last ending' task in Cand_l; that is, it is the task corresponding to the rightmost ending interval in Cand_l. If l is a 1-level, set $\text{Cand2}_l = \text{Cand}_l - \{\text{last}(\text{Cand}_l)\}$; otherwise $\text{Cand2}_l = \text{Cand}_l$. For each 1-level l, last(Cand_l) is the task that will do the actual jump from level l in the optimal schedule to be constructed.

Step 4 For each level l, determine NumTo_l, the number of levels that jump to level l in an LMJ schedule, which is the number of levels that jump to level 0 in G_{l+1} minus the number of levels higher than l that jump to level 0 in G_l. Notice that if l is a 1-level then the number of tasks on level l that remain to be scheduled when processing is to start for level l is $2(\text{opt}(G_l) - \text{opt}(G_{l+1})) - 1$. Hence,

if l is a 1-level
then $\text{NumTo}_l = |\text{level } l| - 2(\text{opt}(G_l) - \text{opt}(G_{l+1})) + 1$
else $\text{NumTo}_l = |\text{level } l| - 2(\text{opt}(G_l) - \text{opt}(G_{l+1}))$.

Step 5 Let Used_l be the set of tasks from level l that are used in jumps to level l. If

$X_1 = \textit{level}_l - \text{Cand}_l$ /* X_1 is a subset of tasks on level l used in jumps to level l */
$y = \text{NumTo}_l - |X_1|$ /* y more tasks that are in jumps to level l have to be */
/* selected from Cand2_l */
$X_2 = \{y$ earliest starting tasks from $\text{Cand2}_l\}$

then $\text{Used}_l = X_1 \cup X_2$. Set $\text{Cand3}_l = \text{Cand2}_l - X_2$.

Step 6 Find a maximum matching for the bipartite graph $BG = (A, B, E)$ where

$A = \{c \mid \exists$ a 1-level $l : c = \text{last}(\text{Cand}_l)\}$

$B = \{d \mid \exists$ level $l : d \in \text{Used}_l\}$

$E = \{\langle c, d\rangle \mid c \in A, d \in B, c$ and d overlap$\}$.

(Note that we will show the size of the maximum matching to be equal to the size of B.)

Step 7 An optimal schedule can now be constructed. First we construct a sequence $Seq[1, \ldots, 2*\text{opt}(G)]$ as follows. For each level l, the elements of Cand3_l appear in $Seq[2*\text{opt}(G_{l+1})+1, \ldots, 2*\text{opt}(G_{l+1})+1+|\text{Cand3}_l|]$. If l is a 1-level then last(Cand_l) is placed in $Seq[2*\text{opt}(G_{l+1})+2+|\text{Cand3}_l|]$. For every matching $\langle c, d\rangle$ found in Step 6, if c is placed in $Seq[i]$ then d is placed in $Seq[i+1]$. Now construct a schedule from Seq by scheduling the tasks that appear in $Seq[2i-1]$, $Seq[2i]$ at time i.

5.2 Special Bipartite Graph Matching

Before proving that Algorithm MM generates an optimal schedule, we consider the problem of finding a maximum matching on the bipartite graph constructed in Step 6 of Algorithm MM.

Definition 3: A convex bipartite graph $G = (A, B, E)$ is a bipartite graph where A and B are sequences of vertices $\langle a_1, \ldots, a_{m_A} \rangle$ and $\langle b_1, \ldots, b_{m_B} \rangle$ respectively. $(a, b) \in E$ implies $a \in A$ and $b \in B$ and $(a_j, b_i) \in E$ iff $f_i \leq j \leq l_i$. (That is, f_i (l_i) is the index of the first (last) element in A to which b_i has an edge).

Definition 4: The lexicographic first matching on a convex bipartite graph G, LFM(G), is defined recursively as follows: if a_1 is not connected to any vertex in B then LFM(G) = LFM($(\langle a_2, \ldots, a_{m_A} \rangle, \langle b_1, \ldots, b_n \rangle, E)$), else LFM($G$) = $(a_1, b_i) \cup$ LFM($\langle a_2, \ldots, a_{m_A} \rangle, \langle b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_{m_B} \rangle, E'$)) where $(a_1, b_i) \in E$ and for $1 \leq j < i$, $(a_1, b_j) \notin E$ and $E' = \{(x, y) \mid (x, y) \in E$ and $x \neq a_1$ and $y \neq b_i\}$. Note that LFM(G) is unique.

Definition 5: The *shell representation* of a convex bipartite graph G is the ordered list of tuples $B = \langle [f_1, l_1], \ldots, [f_{m_B}, l_{m_B}] \rangle$ such that for $1 \leq i \leq k, a_{f_i}(a_{l_i})$ is the lowest (highest) indexed vertex in A which has an edge to b_i. The leading k-shell is the sublist $B_k = \langle [f_1, l_1], \ldots, [f_k, l_k] \rangle$.

If all vertices in list B of a convex bipartite graph G are matched in the LFM(G) then we call G a *complete* convex bipartite graph. Throughout the rest of this section we will assume G is a shell representation of a complete convex bipartite graph.

Theorem 5: LFM(G) can be found in $O(\log n)$ time using $O(n^3/\log n)$ processors on the CREW PRAM model.

After some definitions and examples, we will prove Theorem 5 by constructing an appropriate algorithm. Throughout this section we say a_i and b_j are matched if $(a_i, b_j) \in$ LFM(G).

Definition 6: In G, the pair (a_i, b_j) is a *candidate match pair* if $(a_i, b_j) \in E$ and for all $k < i$, b_j is not matched with a_k in LFM(G).

The algorithm we are going to construct will identify the actual matches by first identifying candidate matches. We motivate the selection of candidate matches with an example.

Example: Consider the graph $G = (A, B, E)$ partially shown in Figure 2. The convex edge set for each b_i is shown as a shaded region bounded by edges (a_{f_i}, b_i) and (a_{l_i}, b_i); thus the dashed edges denote the shell edge set $\{(a_{f_i}, b_i) \mid 1 \leq i \leq m_B\}$.

In this example, $B_5 = \langle [1, 3], [3, 3], [3, 4], [3, 4], [4, 5] \rangle$. Suppose we are trying to determine whether b_5 can be a candidate match for a_5. We must verify that $a_1, \ldots, a_4$ will not match with b_5 in LFM(G). It is straightforward that in LFM(G), a_1 matches with b_1; a_2 does not match with any vertex in B_5; a_3 matches with b_2; a_4 matches with b_3; and hence b_5 is a candidate match for a_5. (Note that b_5 would also be a candidate match for a_4.)

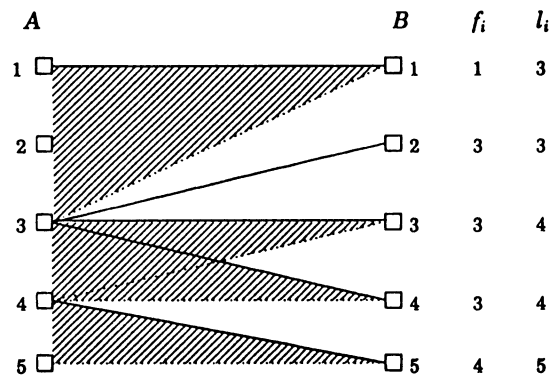


Figure 2: Example

The key thing here is that B_5 contains set $I = \langle b_2, b_3, b_5 \rangle$ which have edges only with a_3, a_4 and a_5; and a_3 can match with $b_2 = [3,3]$, a_4 can match with $b_3 = [3,4]$, and $b_5 = [4,5]$ is then a candidate match for a_5. Note that in this case the candidate matches are the actual matches in LFM(G); this is not a necessary condition: the set $I = \langle b_3, b_4, b_5 \rangle$ serves equally in verifying that b_5 can match with a_5.

We find that b_j is a candidate match for a_i

$\Leftrightarrow$ b_j is not a match for any $a_k, 1 \leq k < i$ in LFM(G)

$\Leftrightarrow$ b_j is not a match for any $a_k, f_j \leq k < i$

$\Leftrightarrow$ all $a_k, f_j \leq k < i$ should have a match in $\langle b_1, \ldots, b_{j-1} \rangle$.

Restating this, b_j is a candidate match for a_i provided:

Cond: in LFM(G), all vertices in $\langle a_{f_j}, \ldots, a_{i-1} \rangle$ have a match in $B_{j-1} = \langle b_1, \ldots, b_{j-1} \rangle$.

Definition 7: For any vertex $a_i, 1 \leq i \leq m$, the *first candidate match pair* is (a_i, b_j) such that (a_i, b_j) is a candidate match pair and for $1 \leq k < j, (a_i, b_k)$ is not a candidate match pair.

Lemma 1: LFM(G) is the set $\{(a_i, b_j) \mid a_i$ has a candidate match and (a_i, b_j) is the first candidate match pair of $a_i\}$.

proof: Consider any a_i having a candidate match and let (a_i, b_j) be its first-candidate match pair. By definition of candidate match pair, $\forall k < i$, b_j is not matched with a_k in LFM(G). Since the pair is a first-candidate match pair, a_i does not match with any $b_l, l < j$. So $(a_i, b_j) \in$ LFM(G). □

We can find LFM(G) by identifying all candidate pairs using a test for *Cond* and then choosing first candidate pairs. The following lemma provides a test for *Cond*.

Lemma 2: In a complete convex bipartite graph G, a_k can match in B_h iff there exists a list $I \subseteq B_h, I = \langle [f_{\sigma(1)}, l_{\sigma(1)}], \ldots, [f_{\sigma(i)}, l_{\sigma(i)}] \rangle$ (with same ordering as B_h) where $|I| = i$ and $\forall j, 1 \leq j \leq i, k-i+1 \leq f_{\sigma(j)} \leq k$.

proof: [⇒] Given a complete convex bipartite graph G, vertex a_k and list B_h such that a_k matches in B_h, show that there exists a list I having the properties described above.

Let K be the largest contiguous list of a's ending with a_k such that all elements in K have their LFM(G) matches in B_h. Let I be the list of matches for elements in K; then $I \subseteq B_h$ and $|K| = |I|$. Clearly, $\forall\, b_j \in I, f_j \leq k$ else b_j could not match with any element in K. Suppose $\exists\, b_p \in I$ such that $f_p < k - |I| + 1$. We know b_p matches with some $a_j \in K$, so by definition of candidate match, all vertices in $\langle a_{f_p}, \ldots, a_{k-|I|} \rangle$ must have matches in $B_{p-1} \subset B_h$. This contradicts K being maximal. So the list I has the desired properties.

[⇐] Given a complete convex bipartite graph G with distinguished vertex a_k, list B_h, and *smallest* list $I \subseteq B_h$ with above properties. Show that a_k can match in B_h (by induction on h).

For $h = 1$: $B_1 = \langle b_1 \rangle$, so $I = \langle b_1 \rangle = \langle [k, l_1] \rangle, l_1 \geq k$. So a_k can match with $b_1 \in B_1$ by definition of candidate match pair.

Assume true for $h' < h$: Since I is the smallest sublist of B_h with the above properties, there must be an element in I adjacent to a_{k-i+1} (if not then for $1 \leq j \leq i, k - i + 2 \leq f_{\sigma(j)} \leq k$ and a smaller I could have been chosen, contradicting the assumption that I was smallest). Let b_q be the lowest-indexed element in I adjacent to a_{k-i+1}. Now consider I' formed from $I - \langle b_q \rangle$ by replacing all vertices of the form $[k - i + 1, l]$ with $[k - i + 2, l]$. Note that all vertices in I' still have incident edges, since all l's other than l_q must be greater than $k - i + 1$ (else G would not be complete). Let $B'_h = B_h - \langle b_q \rangle$ and let G' be the subgraph of G induced by I'.

So, $|I'| = |I| - 1 = i - 1$ and $\forall\, j, 1 \leq j \leq i', k - i' + 1 \leq f_{\sigma(j)} \leq k$. Since $I' \subseteq B'_h$, by the induction hypothesis a_k can match in B'_h in G'. But this remains true in G as well, since a_{k-i+1} has b_q as its candidate match independent of any other candidate match pairs in G. That is, any matching in G' which verifies that a_k can match in B'_h will remain unchanged in G and thus serve to verify that a_k can match in B_h in G. □

Proof of Theorem 5: Let $n = m_A + m_B$. We first solve the problem of calculating $C(i, k, h)$ = number of elements in B_h whose first component is in the range $[i, k]$ for $1 \leq i \leq k \leq m_A, 1 \leq h \leq m_B$. We can solve $C(1, k, h), 1 \leq h \leq m_B$ using standard techniques for parallel prefix in $O(\log n)$ time using $O(n/\log n)$ processors. So $C(1, k, h), 1 \leq k \leq m_A, 1 \leq h \leq m_B$ can be obtained in $O(\log n)$ time using $O(n^2/\log n)$ processors. Since $C(i, k, h) = C(1, k, h) - C(1, i - 1, h)$, all $C(i, k, h)$ can be done in $O(\log n)$ time using $O(n^3/\log n)$ processors. Note, a_k can match in B_h iff $\exists\, i : C(i, k, h) \geq k - i + 1$ by Lemma 2. Now, $(a_k, b_h) \in$ LFM(G) iff h is the smallest index such that a_k can match in B_h. So, the total cost of constructing LFM(G) is $O(\log n)$ time using $O(n^3/\log n)$ processors. □

We can now state our main result:

Theorem 6: Algorithm MM constructs a maximum matching for $G \in \mathrm{DAG_I}$ in $O(\log^2 n)$ time using $O(n^6/\log n)$ processors.

proof: We proceed by showing that the steps of Algorithm MM do what is specified along with time and processor requirements for each step.

Step 1 Note, $\mathrm{opt}(G_l) = \mathrm{opt}(G'_l)$ iff l is a 1-level. The cost is $O(\log^2 n)$ time using $O(n^6/\log n)$ processors.

Step 2 It is obvious that Cand_l can be determined as specified; the cost is $O(\log^2 n)$ time using $O(n^6/\log n)$ processors.

Step 3 Consider any 1-level l; it follows that l will jump to a lower level (including possibly level 0). Since a sequential greedy algorithm based on right endpoints of intervals generates an optimal schedule, it follows that the task from level l to be done last can be chosen to be the rightmost ending interval in $Cand_l$. The cost is $O(\log n)$ time using $O(n^2)$ processors.

Step 4 NumTo_l can be calculated as specified in $O(\log^2 n)$ time using $O(n^6/\log n)$ processors.

Step 5 Note that a task from level l does not appear in Cand_l iff it is required for a jump from a higher level in every optimal schedule. So X_1 is a subset of tasks from level l that is used for jumps from higher levels. (X_1 corresponds to the set of non-free tasks at a level in Gabow [7].) If $\mathrm{NumTo}_l < |X_1|$, then it means that for some levels that jump to level l, there is more than one choice of what jump can be made in an optimal schedule. Since the vertices correspond to intervals, it follows that for t_1, t_2 on level l, if the interval corresponding to t_1 starts earlier than the interval corresponding to t_2, then the set of intervals at a higher level that can match with t_1 contains the set of intervals at a higher level that can match with t_2. Hence we can pick an appropriate number of earliest starting intervals from $\mathrm{Cand2}_l$ to be intervals that are used in jumps to level l. The cost is $O(\log n)$ time using $O(n^2)$ processors.

Step 6 In $BG = (A, B, E)$, if the vertices in A and B are sorted on right endpoint then it is easy to see that BG is a complete convex bipartite graph. From the construction it follows that $\mathrm{LFM}(BG)$ will give us all the jumps in an optimal schedule. To get the shell representation of BG, first determine set $B' = \langle [f_{\alpha(1)}, l_{\alpha(1)}], \ldots, [f_{\alpha(n)}, l_{\alpha(n)}] \rangle$ in $O(\log n)$ time using $O(n)$ processors as follows: list A is a sorted sequence of real intervals and by the convexity property on B, f_j and l_j can be found by binary search on A. Sort B' to obtain list $B = \langle [f_1, l_1], \ldots, [f_n, l_n] \rangle$ in $O(\log n)$ time using $O(n)$ processors. Now from Theorem 5, it follows that $\mathrm{LFM}(BG)$ can be determined in $O(\log n)$ time using $O(n^3/\log n)$ processors.

Step 7 It is easy to see that the intervals in a level form a clique and hence the elements of $\mathrm{Cand3}_l$ can appear in any order. If all the jumps in an optimal schedule are known then the actual optimal

schedule can be constructed as described since we restrict ourselves to a level-first schedule. The cost is constant time using $O(n)$ processors.

Hence an optimal schedule can be constructed in $O(\log^2 n)$ time using $O(n^6/\log n)$ processors. □

Corollary 1: A maximum matching on interval graph G can be found in $O(\log^2 n)$ time using $O(n^6/\log n)$ processors on the CREW PRAM model.

proof: Given an interval graph G, construct an interval representation for G in $O(\log n)$ time using $O(n^3)$ processors on the CRCW PRAM model by the algorithm of Novick [14].

Next, form $G' \in \text{DAG}_\text{I}$ from G. This can be done in $O(\log n)$ time using $O(n^2)$ processors. A maximum matching on G corresponds to an optimal two-processor schedule on G' which can be found in $O(\log^2 n)$ time using $O(n^6/\log n)$ processors.

Thus the total time is $O(\log^2 n)$ using $O(n^6/\log n)$ processors on the CREW PRAM model. □

6 Other Parallel Algorithms on Interval Graphs

We now show that the algorithm for maximum matching on interval graphs can be improved if we are working with representations of proper interval graphs. In Section 6.1 we give an algorithm for DFS on representations of interval graphs and in Section 6.2 we show how the DFS algorithm can be used to find a maximum matching on representations of proper interval graphs.

6.1 Depth-First Search

In this section we show that a depth-first search (DFS) tree of an interval graph representation can be obtained in $O(\log n)$ time using $O(n^2/\log n)$ processors.

Given an interval set I, let the intervals be sorted by right endpoints so that $b_i \leq b_{i+1}$ for $1 \leq i < n$. We define $last(I) = j$, such that $b_j = \max\{b_k\}$. For each interval $i \in I$, we define $parent(i) = j$, if $b_j = \min\{b_k \mid k > i, a_k \leq b_i \leq b_k\}, = 0$, otherwise. $parent(i)$ is the interval overlapping i which ends first after the end of i, if it exists. Given a connected interval graph G, construct $T = (V, E')$ where $E' = \{(i, parent(i)) \mid parent(i) \neq 0\}$.

Theorem 7: For an interval graph G, T (as described above) is a DFS tree and can be constructed in $O(\log n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model.

proof: We show that T is a spanning tree with no cross edges. Note that the root of T is $last(I)$. For each vertex i (other than the root), $parent(i)$ is the parent of i in T.

1. Suppose there exists interval $i, i \neq n$, such that $parent(i) = 0$. Then i does not intersect any intervals in the set $S = \{i+1, i+2, \ldots, n\}$. Additionally, no interval ending before b_i can intersect an interval in S. So there can be no edge between the sets S and $I - S$. This contradicts G connected. Hence, $parent(i)$ exists for every interval other than n; i.e., every vertex other than the root has a parent. Since each edge in T joins an interval to one ending later, there can be no cycles. Since every vertex other than the root contributes an edge, there are $n-1$ edges. Thus T is a spanning tree.

2. Suppose some edge $e = (i,j)$ in $E - E'$ is a cross edge. Then $[a_i, b_i]$ overlaps $[a_j, b_j]$. Without loss of generality, assume $b_i \leq b_j$. Let k be the least common ancestor of i and j in T. Then $b_i \leq b_k$ and $b_j \leq b_k$. Let $i, i_1, i_2, \ldots, i_m, k$ be the vertices on the path between i and k. Then $b_i \leq b_{i_1} \leq b_{i_2} \leq \ldots \leq b_{i_m} \leq b_k$. Since $parent(i) = i_1, b_{i_1} \leq b_j$. Since (i,j) is an edge, intervals i and j overlap; so i_1 and j overlap. Now (i_1, j) is a cross edge. Repeating this argument for each vertex on the path from i to k results in $b_k \leq b_j$, a contradiction. Thus (i,j) cannot be a cross edge.

The time to compute the above DFS tree for a connected graph G is simply the time to compute $parent(i)$ for all i in I. For any i, $parent(i)$ can be calculated in $O(\log n)$ time using $O(n/\log n)$ processors by simply finding the minimum right endpoint of all intervals that overlap with i and end after i. The total cost of computing the DFS is then $O(\log n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model. □

6.2 Maximum Matching in Proper Interval Graphs

In the case of proper interval graphs, maximum matching is easily solved via depth-first search.

Theorem 8: In a proper interval graph G, a maximum matching can be found in $O(\log n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model.

proof: Assume G is connected. Let set I be sorted by right endpoints. Since G is proper, each interval i except $last(I)$ intersects $i+1$. Hence the edges of the DFS tree are $E = \{(i, i+1) \mid 1 \leq i < n\}$. Clearly, these edges form a path of length $n-1$ and there are $\lfloor \frac{n}{2} \rfloor$ odd edges on this path. The odd edges form a matching, and this matching is maximum since no matching on any graph of n vertices can exceed size $\lfloor \frac{n}{2} \rfloor$. Now consider a proper interval graph G having more than one connected component. Applying DFS results in a unary spanning tree of each component. By a simple doubling strategy, we can locate the odd edges in all of these spanning trees; these edges form maximum matchings on the connected components. Since the components are independent, these edges form a maximum matching on the entire graph.

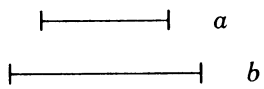
Thus, maximum matching on representations of proper interval graphs can be obtained in $O(\log n)$ time using $O(n^2/\log n)$ processors. □

7 PT-Optimal Parallel Algorithms on Interval Graphs

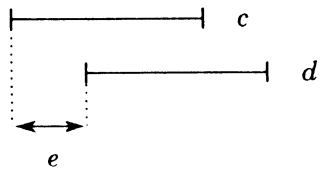
Bertossi and Bonuccelli [2] give several parallel algorithms for interval graphs that rely on parallel computation of *right*(i) and *next*(i) for each interval i. Assume I is sorted by increasing right endpoints; then *right*(i) is the rightmost ending interval overlapping interval i, that is, *right*(i) $= j$ such that $b_j = \max\{b_k \mid i < k, a_k < b_i \leq b_k\}$, $= 0$ otherwise (this definition is slightly different from that in [2]). Note that the definition requires I sorted so as to break ties consistently. *next*(i) is the leftmost ending interval beginning after the end of interval i, that is, *next*(i) $= j$ such that $b_j = \min\{b_k \mid b_i < a_k\}$, $= 0$ otherwise. Bertossi and Bonuccelli give algorithms for *right*(i) and *next*(i) (for all i) which take $O(\log n)$ time and $O(n^2/\log n)$ processors. In Section 7.1, we show that *right*(i) and *next*(i) for all i can be computed in $O(\log n)$ time with $O(n)$ processors (note that these computations are PT-optimal since they are equivalent to sorting). In Section 7.2 we use *right*(i) and *next*(i) to develop several PT-optimal algorithms for representations of interval graphs.

7.1 Faster Computation of *right*(i) and *next*(i)

To motivate our algorithm for computing *right*(i), consider an interval a properly contained in some interval b.



Clearly, no interval i will choose *right*(i) $= a$ since if i overlaps a then i also overlaps b, and b ends right of a. After discarding intervals properly contained in another interval, we are left with a proper interval graph. In the example below, any interval ending in e will not choose d but may choose c.



To compute *right*(i) for all i, perform the following steps:

Step 1 Form set P from I by discarding properly contained intervals in I, i.e., P is a smallest sized subset of I such that for all $i \in I$, there exists $j \in P$ such that j contains i.

Step 2 Sort the left endpoints of intervals in P to obtain $P' = \langle a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(h)} \rangle$. The crucial property of intervals in P is that for any $i, j \in P, i \neq j$, if interval i starts before interval j, then interval i ends before interval j. Therefore, for $i \in I$, if *right*(i) $\neq 0$ then *right*(i) $= \sigma(k)$

where i ends in the interval $[a_{\sigma(k)}, a_{\sigma(k+1)})$. Since the intervals $[a_{\sigma(k)}, a_{\sigma(k+1)})$ are sorted and nonoverlapping, the search can be done in $O(\log n)$ time for each interval. The total cost of this step is $O(\log n)$ time, $O(n)$ processors.

We must now consider how to implement Step 1 efficiently in parallel.

1. Let $E = \{(a, l) \mid \exists k \in I : a_k = a\} \cup \{(b, r) \mid k \in I : b_k = b\}$, the set of left and right endpoints of intervals in I. Sort the elements of E on the first component, forming list L_{lr}. Form list L_l from list I by sorting elements of I by increasing left endpoints, and then sorting by decreasing right endpoints all subsequences of elements having the same left endpoint. Note that elements in L_l may have the same left and/or right endpoints; for this reason, we maintain a mapping onto I so that the corresponding vertices are known. Both lists may be formed in $O(\log n)$ time using $O(n)$ processors.

2. Construct a list $L_{l'}$ from L_l by replacing each element in L_l by the index of its right endpoint in L_{lr}. This can be done by binary search in $O(\log n)$ time using $O(n)$ processors.

3. Now, for any interval $k \in I$, we can determine whether it is properly contained within some other interval of I as follows: Let a_k appear as the j^{th} element in L_l and let the j^{th} element of $L_{l'}$ be p. k is properly contained within some other interval of I iff $\exists\, j', j' < j$ and $L_{l'}[j'] \geq p$. (If such a j' exists, then there exists an interval that starts no later than k since $j' < j$, and that interval ends not before interval k since $L_{l'}[j'] \geq p$.) So, perform a parallel prefix computation to obtain the largest value in every prefix of $L_{l'}$. Now for each $2 \leq i \leq n$, the interval corresponding to $L_{l'}[i]$ is to be discarded iff $\max\{L_{l'}[1 \ldots i-1]\} \geq L_{l'}[i]$. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors.

Similarly, we can identify all intervals that cover another interval. (Here, we would perform a parallel prefix computation to obtain the smallest value in every suffix of $L_{l'}$.)

As with *right*(i), *next*(i) may be computed more efficiently if we first discard some intervals. If interval b properly contains a, then a will always be chosen over b for *next*(i) since a ends left of b. After discarding all intervals that cover another interval, we again are left with a proper interval graph. To compute *next*(i), perform the following steps:

Step 1 Form set P from I by discarding intervals that properly contain other intervals from I; i.e., P is the smallest sized subset of I such that for all $i \in I - P$ there exists $j \in P$ such that i contains j.

Step 2 Sort the left endpoints of intervals in P, and denote this sorted list by $P' = \langle a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(h)} \rangle$. The crucial property of intervals in P is that for any $i, j \in P, i \neq j$, if interval i starts before interval j, then interval i ends before interval j. Therefore, for $i \in I$, if *next*(i) $\neq 0$ then *next*(i) $= \sigma(k+1)$ where i ends in the interval $[a_{\sigma(k)}, a_{\sigma(k+1)})$. Since the intervals $[a_{\sigma(k)}, a_{\sigma(k+1)})$

are sorted and nonoverlapping, the search can be done in $O(\log n)$ time for each interval. The total cost of Step 2 is $O(\log n)$ time, $O(n)$ processors.

We have seen that Step 1 can be computed in $O(\log n)$ time using $O(n)$ processors, so both *right*(i) and *next*(i) for all i can be computed in $O(\log n)$ time using $O(n)$ processors.

7.2 Several PT-optimal Algorithms for Interval Graphs

In this section we present parallel algorithms for maximum weighted clique, maximum independent set, minimum clique cover, and minimum cardinality dominating set for interval graphs; and Hamiltonian circuit and minimum bandwidth for proper interval graphs. Each of these is an improvement over the corresponding algorithm in Bertossi and Bonuccelli [2]. Further, all of these algorithms except minimum bandwidth are PT-optimal.

Given an interval graph with weights associated with the vertices, we now describe how to find a *maximum weighted clique*. First of all we can restrict ourselves to considering the case where all the weights are positive. This is because if all the weights are non-positive then the maximum weighted clique consists simply of a single vertex with maximum weight. Otherwise, no non-positive vertex weight can be part of a maximum weighted clique and hence all the vertices with non-positive weights can be dropped.

With each point p on the real line we can define its weight to be the sum of the weights of all the intervals that contain the point p. To find maximum weighted clique it is enough to find a point on the real line with maximum weight. The weights of points on the real line change only when either an interval starts or an interval ends and hence maximum weighted clique can be found as follows. Sort the $2n$ points $a_1, \ldots a_n, b_1, \ldots, b_n$ in increasing order and assign positive weights to left endpoints and negative weights to right endpoints. The clique weights can be found by computing the partial sum and then the maximum weighted clique can be found. The sorting can be done in $O(\log n)$ time using $O(n)$ processors [5]; the partial sums can be obtained in $O(\log n)$ time using $O(n)$ processors [13]. The maximum weighted clique can, therefore, be found in $O(\log n)$ time using $O(n)$ processors.

An *independent set* of a graph is a subset of its vertices such that no two vertices in the subset are connected by an edge. For interval graphs a maximum cardinality independent set S can be defined inductively as: *first*$(I) \in S$; if $i \in S$, then *next*$(i) \in S$. So if *next*(i) has been computed for all $i \in I$ then S can be obtained by the standard parallel doubling technique. The entire computation requires $O(\log n)$ time using $O(n)$ processors.

In the *minimum clique cover* problem, a minimum cardinality partitioning of the nodes of the graph into cliques is required. For interval graphs this can be done by augmenting its maximum independent set. If $S = \{i_1, \ldots, i_m\}$ is a maximum independent set, then there exists a minimum clique cover $C_{i_1}, \ldots, C_{i_m}$ such

that interval $i_j \in S$ belongs to C_{i_j} and each $h \notin S$ belongs to C_{i_k} iff

$$b_{i_k} = \min\{b_i \mid i \in S \text{ and } a_h < b_i \leq b_h\}$$

For each vertex h, the clique to which it belongs can be found in $O(\log n)$ time by doing binary search on S sorted by right endpoints. The minimum clique cover can therefore be found in $O(\log n)$ time using $O(n)$ processors.

A *dominating set* for a graph is a subset D of its vertices such that every vertex of the graph is either in D or is connected by an edge to a vertex in D. The sequential greedy algorithm of Booth and Johnson [3] finds a minimum cardinality dominating set for interval graphs as follows: $right(first(I)) \in D$; if $i \in D$, then $right\ (next(i)) \in D$ where $right(0) = 0$. The standard doubling technique leads to an $O(\log n)$ time, $O(n)$ processor algorithm for minimum cardinality dominating set.

The algorithms given here for Maximum weighted clique, maximum independent set, minimum clique cover and minimum cardinality dominating set can be shown to be PT-optimal as follows. A sequential lower time bound of $\Omega(n \log n)$ can be shown for the problem of determining whether n real line intervals are pairwise disjoint [16]. Gupta, Lee and Leung [9] have shown a sequential lower time bound of $\Omega(n \log n)$ for maximum weighted clique, maximum independent set, and minimum clique cover on interval graphs (with representation given) by arguing that a solution to any one of these problems also solves the problem of determining whether n real line intervals are pairwise disjoint. A similar argument applies to minimum cardinality dominating set.

A graph has a *Hamiltonian circuit* if there is a circuit in the graph where each vertex of the graph appears exactly once. In Bertossi [1] it is shown that a Hamiltonian circuit exists for an interval graph if and only if intervals in $I - \{first(I)\}$ can be partitioned into disjoint subsets J and K such that:

(1) $right(first(I)) \in J$, and if $j \in J$, then $right(j) \in J$, and this process should result in $last(I) \in J$.

(2) the subgraph represented by $K \cup \{first(I), last(I)\}$ has a Hamiltonian path.

Further, Bertossi [1] has shown that for proper interval graphs, a Hamiltonian path exists if and only if consecutive intervals overlap when the intervals are sorted on left endpoints. The partitioning of the intervals can be done in $O(\log n)$ time using $O(n)$ processors. The two conditions on this partitioning can also be checked within these bounds. If the partitioning can be done, then the Hamiltonian circuit is formed by putting together the Hamiltonian path on $K \cup \{first(I), last(I)\}$ with the path formed by sorting elements of J on right endpoints. The entire computation requires $O(\log n)$ time using $O(n)$ processors, which is PT-optimal, as Weide [17] has shown an $\Omega(n \log n)$ lower bound for sequential algorithms for Hamiltonian circuit on proper interval graphs.

A graph $G = (V, E)$ has a *bandwidth* h if there is a function $f : V \rightarrow 1, \ldots, |V|$ such that for all $(u, v) \in E$ $|f(u) - f(v)| \leq h$. The *minimum bandwidth* problem is to find the smallest such h. For $1 \leq i \leq n$, let an *overlap clique* C_i of I be a maximal set of intervals all overlapping at a_i. For proper interval graphs, Orlin,

Bonuccelli, and Bovet [15] have shown that if the intervals are indexed by increasing right endpoint then the intervals within each overlap clique C_i are consecutive, that is, $C_i = \{j, j+1, \ldots, k\}$ for some j and k. From this it follows that the minimum bandwidth is one less than the size of the largest overlap clique. Since a maximum clique is an overlap clique, the bandwidth for proper interval graphs can be obtained as a byproduct of finding the maximum clique in $O(\log n)$ time using $O(n)$ processors.

Recently, Klein [11] has given $O(\log^2 n)$ time, $O(n+m)$ processor algorithms on the CRCW PRAM model for the following interval graph problems: recognition, maximum weighted clique, maximum independent set, optimal coloring, breadth-first search, depth-first search, and graph isomorphism. Note that these algorithms assume an adjacency list representation and thus have different sequential lower bounds than algorithms requiring an interval representation.

The following tables summarize our results for several parallel algorithms for interval graphs.

PT-optimal Parallel Algorithms for Interval Graph Representation

	Bertossi and Bonuccelli [2]		*present work*	
Algorithm	*time*	*processors*	*time*	*proc*
maximum weighted clique	$O(\log n)$	$O(n^2/\log n)$	$O(\log n)$	$O(n)$
maximum independent set	$O(\log n)$	$O(n^2/\log n)$	$O(\log n)$	$O(n)$
minimum clique cover	$O(\log n)$	$O(n^2/\log n)$	$O(\log n)$	$O(n)$
minimum dominating set	$O(\log n)$	$O(n^2/\log n)$	$O(\log n)$	$O(n)$
Hamiltonian circuit of proper interval graph	$O(\log n)$	$O(n^2/\log n)$	$O(\log n)$	$O(n)$

Other Parallel Algorithms for Interval Graph Representation

Algorithm	*time*	*processors*
minimum bandwidth of proper interval graph	$O(\log n)$	$O(n)$
depth-first search	$O(\log n)$	$O(n^2/\log n)$
maximum matching on proper interval graphs	$O(\log n)$	$O(n^2/\log n)$
maximum matching	$O(\log^2 n)$	$O(n^6/\log n)$

References

[1] A. A. Bertossi. Finding hamiltonian circuits in proper interval graphs. *Information Processing Letters 17*, pages 97–101, 1983.

[2] A. A. Bertossi and M. A. Bonuccelli. Some parallel algorithms on interval graphs. *Discrete Applied Mathematics 16*, pages 101–111, 1987.

[3] K. Booth and J. H. Johnson. Dominating sets in chordal graphs. *SIAM Journal of Computing 11*, pages 191–199, 1982.

[4] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica 1*, pages 200–213, 1972.

[5] R. Cole. Parallel merge sort. In *27th FOCS*, pages 511–516, 1986.

[6] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal of Applied Mathematics*, 17(4):784–789, 1969. Erratum, SIAM J. Appl. Math. 20 (1971), 141.

[7] H. N. Gabow. An almost-linear algorithm for two-processor scheduling. *Journal of the Association for Computing Machinery*, 29(3):766–780, July 1982.

[8] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Computer Science and Applied Mathematics. Academic Press, Inc., 1980.

[9] U. I. Gupta, D. T. Lee, and Y. T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12:459–467, 1982.

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

[11] P. Klein. Efficient parallel algorithms for chordal graphs. Laboratory for Computer Science, MIT, 1988.

[12] D. Kozen, U. V. Vazirani, and V. V. Vazirani. $\mathcal{NC}$ algorithms for comparability graphs, interval graphs, and unique perfect matchings. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 496–503, December 1985. in LNCS, vol. 206, Springer-Verlag.

[13] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers, C-34*, pages 965–968, 1985.

[14] Mark Novick. Personal communications.

[15] J. B. Orlin, M. A. Bonuccelli, and D. P. Bovet. An $O(n^2)$ algorithm for coloring proper circular arc graphs. *SIAM Journal of Algebraic Discrete Methods*, 2:88–93, 1981.

[16] F. P. Preparata and M. I. Shamos. Computational geometry: An introduction. In *Texts and Monographs in Computer Science.* Springer-Verlag, 1985.

[17] B. Weide. A survey of analysis techniques for discrete algorithms. *Computing Surveys*, 9:291–313, 1977.