Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Parallel Interval Order Recognition and Construction of Interval Representations

Michael A. Bender
Michel Gastaldo
Michel Morvan

September 1993

Research Report N° 93-27

Ecole Normale Supérieure de Lyon

46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

Parallel Interval Order Recognition and Construction of Interval Representations

Michael A. Bender
Michel Gastaldo
Michel Morvan

September 1993

Abstract

Parallel algorithms for recognizing and representing interval orders are proposed for different models of parallel random access machines (PRAM). The algorithms accept as input a transitively-closed directed graph with N nodes and M edges. They run in time $O(\log N)$ with $O(N+M)$ processors and $O(N+M)$ space and in constant time with $O(N^2)$ processors and $O(N^2)$ space depending on the data structure and the PRAM model used. Optimal probabilistic algorithms for PRAM are also presented as well as algorithms for distributed-memory machines.

Keywords: Parallel algorithms, PRAM, partial order, Interval Order.

Résumé

Nous proposons des algorithmes parallèles de reconnaissance et de représentation pour les ordres d'intervalles, fonctionnant sur différents modèles de machine PRAM. Ces algorithmes prennent en entrée un graphe fermé transitivement comptant N nœuds et M arcs. Leur complexité varie de $O(\log N)$ avec $O(N+M)$ processeurs et un espace en $O(N+M)$ à une constante avec $O(N^2)$ processeurs et un espace en $O(N^2)$, suivant le modèle de PRAM utilisé et la structure de données retenue. Nous présentons également des algorithmes probabilistiquement optimaux sur les PRAM, ainsi que des variantes pour les architectures à mémoire distribuée.

Mots-clés: Algorithmes parallèles, PRAM, Ordres partiels, Ordres d'intervalles.

Parallel Interval Order Recognition and Construction of Interval Representations

Michael A. Bender* Michel Gastaldo†
Michel Morvan

LIP
Unité de Recherche Associée 1398 du CNRS
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cedex 07
France
e-mail: {mbender, gastaldo, mmorvan}@lip.ens-lyon.fr

Abstract

Parallel algorithms for recognizing and representing interval orders are proposed for different models of parallel random access machines (PRAM). The algorithms accept as input a transitively-closed directed graph with N nodes and M edges. They run in time $O(\log N)$ with $O(N+M)$ processors and $O(N+M)$ space and in constant time with $O(N^2)$ processors and $O(N^2)$ space depending on the data structure and the PRAM model used. Optimal probabilistic algorithms for PRAM are also presented as well as algorithms for distributed-memory machines.

Key words: Parallel algorithms, recognition algorithms, PRAM, partial order, Interval Order.

1 Introduction

An important problem in the algorithmic study of discrete structures is that of developing efficient recognition algorithms for particular classes. Another important problem is that of finding efficient methods of representing the objects in the classes, so that basic operations can be executed rapidly. In the case of ordered structures, these problems have been studied in depth by, for example, Golumbic [Gol85], Mohring [Moh89], Habib, and Jegou [HJ85], Pnuelli, Lempel, and Even [PLE71], Spinrad [Spi85], Spinrad and Valdes [SV83], M. M. Syslo [Sys82], Valdes, Tarjan, and Lawler [VTL82]. Many different classes of orders have been analyzed both from a theoretical point of view, and also with specific applications in mind. Almost all of the algorithms which have been developed, however, are sequential algorithms. We are interested in developing parallel algorithms to complement the existing sequential algorithms. In this paper we focus on interval orders.

*Supported by a fellowship from the Rotary Foundation.
†Supported by the research program C3, and by the Direction des Recherches Etudes et Techniques.

Interval orders are used to model the disjoint and overlapping structure of a set of intervals of the real line. They have many applications, including modeling gene structure in biology, chronological dating in archeology, preference and indifference relations in measurement theory, existence of species in paleontology, consecutive retrieval, VLSI channel routing, and scheduling in computer science. They have been extensively studied, among others, by Bogart [Bog82], Fishburn [Fis70] [Fis85b] [Fis85a], Mohring [Moh89], and Weiner [Wei14].

The first efficient sequential recognition algorithm was developed by Papadimitriou and Yannakakis in the context of scheduling [PY79]. Given a transitively-closed, directed acyclic graph (*i.e.* an order) with N nodes and M edges as input, the algorithm determines whether the partial order is an interval order. The algorithm is presented in [Moh89] so that, when the graph is an interval order, it returns a corresponding interval representation. The algorithm stores the graph using an adjacency-matrix data structure, and also assumes that the out-degrees of the nodes are known. This sequential algorithm has linear time complexity in the number of nodes and edges (*i.e.* $O(N + M)$ time) and quadratic space complexity in the number of nodes (*i.e.* $O(N^2)$ space).

In the past, interval order recognition algorithms have been proposed by Baldy and Morvan [BM93], by Gabow [Gab81] and Garbe [Gar92]. These algorithms are improvements on the algorithm of [PY79] in two respects. First, they use linear time *and* linear space in the number of nodes and edges. Second, they accept as input any directed acyclic graph—not necessarily transitively-closed—and determine whether the transitive closure of the graph is an interval order. Instead of using an adjacence matrix data structure, these algorithms explicitly store the edges as couples.

Here, we propose parallel algorithms for recognizing interval orders, and parallel algorithms for determining interval representations.

Section 2 introduces the basic definitions and properties of interval orders, which we will exploit in our interval recognition algorithms. In section 3 we propose PRAM parallel interval recognition algorithms including a probabilistically optimal algorithm and a constant time algorithm (under different hypotheses). In section 4 we propose PRAM parallel algorithms for constructing interval representations, also including a probabilistically optimal algorithm and a constant time algorithm. We present parallel recognition and representation algorithms for distributed memory machines in section 5.

2 Definitions and Properties of Interval Orders

Every directed acyclic graph $G = (V, E)$ defines a partial order by making use of the *transitive closure* of G. The transitive closure of graph $G = (V, E)$ is the graph $G_{tc} = (V, E_{tc})$ where

$$(u, v) \in E_{tc} \Leftrightarrow \text{ there exists a path from vertex } u \text{ to vertex } v \text{ in } G.$$

Then $G = (V, E)$ represents partial order $P = (V, <_P)$ if and only if

$$u <_P v \Leftrightarrow (u, v) \in E_{tc}.$$

We introduce the following notation for partial orders and directed acyclic graphs: If $u \in V$ then $Succ(u) = \{v \in V \mid (u, v) \in E\}$ and $TrSucc(u) = \{v \in V \mid (u, v) \in E_{tr}\}$. Furthermore, $Pred(u) = \{v \in V \mid (v, u) \in E\}$ and $TrPred(u) = \{v \in V \mid (v, u) \in E_{tr}\}$ (in the case of ordered sets , $Succ(u) = TrSucc(u)$ and $Pred(u) = TrPred(u)$). An *antichain* of partial order $P = (V, <_P)$ is a subset of V such that the elements are pairwise incomparable. A *maximal antichain* is an antichain which is maximal under inclusion. We denote $A(P)$ the set of antichains of P and and $MA(P)$ set of maximal antichains of P.

A *topological sort* of $G = (V, E)$ is a total ordering of nodes $u_0, u_1, \ldots u_{N-1} \in V$ such that if $i < j$ then $u_i \notin TrSucc(u_j)$. We let $d^-(u_i)$ and $d^+(u_i)$ represent respectively the in-degree of node u_i and out-degree of node u_i. Clearly, $d^+(u_i) = |Succ(u_i)|$ and $d^-(u_i) = |Pred(u_i)|$. Finally, we let N represent the number of nodes of G, and let M represent the number of edges of G.

Let $P = (V, <_P)$ be a *partial order* and let $\mathcal{I}$ be a set of intervals on the real line. A partial order P is an *interval order* if and only if we can associate each element $v \in V$ to an interval $I_v \in \mathcal{I}$ so that

$$u <_P v \Leftrightarrow \forall x \in I_u \, \forall y \in I_v \, , \; x < y \, .$$

That is, $u <_P v$ if and only if interval I_u is entirely to the left of interval I_v on the real line.

Theorem 1 *[Bog82], [Fis70], [Moh89] Let $P = (V, <_P)$ be a partial order. The following statements are equivalent:*

1. *P is an interval order.*
2. *P does not contain four elements u, v, w, and t such that $u < v$ and $w < t$, but $u \not< t$ and $w \not< v$ [See figure 1].*
3. *The successors can be linearly ordered with respect to inclusion. That is, $\forall u, v \in V$ $Succ(u) \subseteq Succ(v)$ or $Succ(v) \subseteq Succ(u)$.*
4. *The predecessors can be linearly ordered with respect to inclusion. That is, $\forall u, v \in V$ $Pred(u) \subseteq Pred(v)$ or $Pred(v) \subseteq Pred(u)$.*
5. *The maximal antichains of P can be linearly ordered, so that for each vertex $v \in V$, the maximal antichains containing v occur simultaneously.*

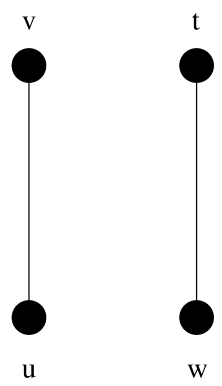

Figure 1: The structure referred to in Theorem 1.2. Partial order P is an interval order if and only if it contains no suborder isomorphic to this structure.

The algorithm of [PY79] indirectly uses statements 3 and 5 as its bases. The algorithm of [BM93], on the other hand, uses the following theorem:

Theorem 2 *[BM93] Let $u_i \in Min(G)$ such that $|Min(G - Min(G)) \cap Succ(u_i)|$ is maximal over $Min(G)$. Then G_{tr} is an interval order if and only if $|Min(G - Min(G)) \cap Succ(u_i)| = |Min(G - Min(G))|$ and $G_{tr} - u_i$ is an interval order.*

3 Parallel Recognition Algorithms

Our parallel recognition algorithms are based on the following two propositions:

Proposition 1 *Let $G = (V, E)$ be a transitively closed acyclic graph. Let $u_0, u_1, \ldots u_{N-1}$ be an ordering of vertices of G such that $d^+(u_i) \geq d^+(u_j)$ for $i < j$. Then this ordering is a topological sort of the nodes.*

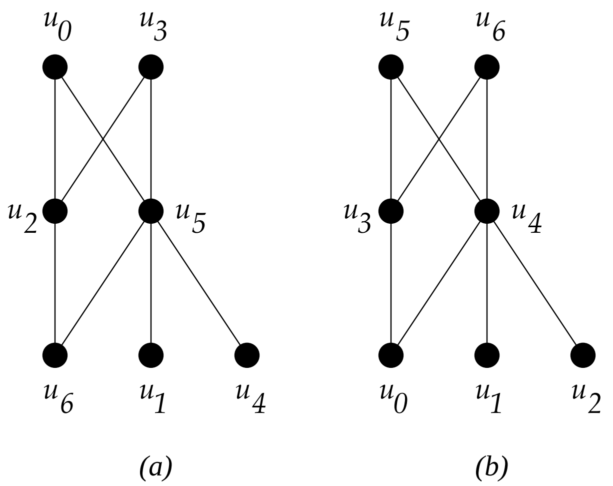

Figure 2: For the sake of presentation, we use the above diagrams to represent transitively-closed directed acyclic graphs. Edge (u_i, u_j) is in the graph if and only if there exists an always ascending path from u_i to u_j. (a) A diagram of a directed acyclic graph. (b) The nodes of the graph represented in diagram-a are relabeled so that the ordering $u_0, u_1, \ldots, u_6$ respects Proposition 1 and is therefore a topological sort.

Proposition 2 *Let $G = (V, E)$ be a transitively closed graph. Let $u_0, u_1, \ldots, u_{N-1}$ be the list of elements of V in topological order, so that $d^+(u_i) \geq d^+(u_j)$ for $i < j$. Then G is an interval order if and only if $Succ(u_{i+1}) \subseteq Succ(u_i)$ for $0 \leq i < N - 1$.*

3.1 Data-Structure Independent Algorithm

In the following we present three parallel algorithms for interval order recognition. The first algorithm employs the *exclusive-read exclusive-write* (EREW) PRAM, where only one processor at a time can read from or write to a memory location. It runs in $O(\log N)$ time with $O(\log N)$ processors and $O(M + N)$ space. The second and third algorithms employ the *priority concurrent-read concurrent-write* (*priority*-CRCW) PRAM. In this model, the processors are numbered, and write conflicts are resolved by having the processor with the smallest index write its value. The second algorithm is probabilistically optimal. It runs in expected time $O(\log N)$ with $O(\frac{N+M}{\log N})$ processors and $O(N^2)$ space. [Note that only $O(N + M)$ of the space will be initialized.] The third algorithm runs in constant time with $O(N^2)$ processors and $O(N^2)$ space.

All the parallel recognition algorithms we propose are based on the following data-structure independent algorithm:

Algorithm 1 (Interval Order Recognition)
Input: A transitively closed graph G.
Output: **True** if G is an interval order and **false** if not.

Step 1: Determine a topological order $u_0, u_1, \ldots u_{N-1}$ such that $d^+(u_{i+1}) \leq d^+(u_i)$.

Step 2: If $Succ(u_{i+1}) \subseteq Succ(u_i)$ for $0 \leq i < N-1$, then G is an interval order, else G is not an interval order.

Theorem 3 *Algorithm 1 determines whether a transitively-closed graph G is an interval order.*

Proof: The proof follows directly from Propositions 1 and 2. □

3.2 An $O(\log N)$ Algorithm

The first parallel interval order recognition algorithm we propose, uses an edge-array data structure, where the edges are explicitly stored as couples in an array. Let $u_0, u_1, \ldots, u_{N-1}$ be the nodes of graph $G = (V, E)$, and let EA denote the edge array used. We choose to relabel each node u_i by u_k where k is u_i's position in the topological sort defined in Proposition 1. After relabeling, nodes $u_0, u_1, \ldots, u_{N-1}$ are in topological order. Note that, because we are storing the edges *explicitly*, if the out-degrees are not supplied, it is easy to calculate them in $O(\log N)$ time using, for example, a sorting and a parallel prefix computation, noted partial sum in the remaining of this paper. During the algorithm, we will store the edges with the following inversed lexicographical order (see figure 3):

$$(u_i, u_j) < (u_k, u_l) \Leftrightarrow \begin{cases} \text{either} & j < l \\ \text{or} & j = l \text{ and } i < k. \end{cases} \tag{1}$$

Note that for every edge (u_i, u_j) in the edge array, when $i \neq 0$, the edge (u_{i-1}, u_j) is stored previously. In the following algorithm whenever a sorting procedure is necessary, we will use the Cole Parallel Merge Sort [Col88]. The Cole Parallel Merge Sort can sort N numbers in time $O(\log N)$ with $O(N)$ processors on EREW PRAM.

$(0,3)$	$(0,4)$	$(1,4)$	$(2,4)$	$(0,5)$	$(1,5)$	$(2,5)$	$(3,5)$	$(4,5)$	$(0,6)$	$(1,6)$	$(2,6)$	$(3,6)$	$(4,6)$

Figure 3: Edge-array structure for the graph represented in figure 2b. The edges are sorted first by ascending second component and then second first component. This data structure is used by Algorithm 2 and Algorithm 5. Note that the graph represented in figure 2b is an interval order, since, for every edge (u_i, u_j) in the edge array, when $i \neq 0$, the edge (u_{i-1}, u_j) is stored previously.

Our algorithm is as follows:

Algorithm 2 Edge-Array Implementation
Input: A transitively-closed graph G represented in edge-array form.
Output: Returns **true** if G is an interval order and **false** if not.

{Implementation of Step 1 of Algorithm 1}
1 Sort the nodes by descending outdegree using the Cole Parallel Merge Sort.
Let $u_0, u_1, \ldots u_{N-1}$ be the nodes in topological order.
2 Sort the edges of array EA according to equation (1) using the Cole Parallel Merge Sort.

{Implementation of Step 2 of Algorithm 1 }
3 **For all** $0 \leq i < M$ **do in parallel**
Let (u_j, u_k) be the edge represented in array position $EA[i]$.
4 **If** $i = 0$ **then**
5 **If** $j = 0$ **then**
6 ReturnValue[i] $\leftarrow$ **true**
7 **Else**
8 ReturnValue[i] $\leftarrow$ **false**
9 **Else if** $j \neq 0$ **then**
10 **If** (u_{j-1}, u_k) is in array position $EA[i-1]$ **then**
11 ReturnValue[i] $\leftarrow$ **true**
12 **Else**
13 ReturnValue[i] $\leftarrow$ **false**
14 Return $\bigwedge_{i=0}^{M}$ ReturnValue[i]

Theorem 4 *Algorithm 2 determines whether graph G is an interval order. It runs on EREW PRAM in $O(\log N)$ time using $O(N+M)$ processors and $O(N+M)$ space.*

Proof: We prove that Algorithm 2 is equivalent to Algorithm 1. Step 1 of Algorithm 2 computes a topological ordering, and therefore, execute step 1 of Algorithm 1. That steps 3-14 of Algorithm 2 execute step 2 of Algorithm 1 follows directly from the definition of edge arrays and from the ordering introduced in equation 1. All sorts and the the conjunction in step 14 can run in $O(\log N)$ time using $O(N+M)$ processors. $\square$

3.3 A Probabilistically Optimal Algorithm

In this section we propose an optimal randomized algorithm. Our algorithm runs on priority PRAM CRCW with an expected time $O(\log N)$ using $O(\frac{N+M}{\log N})$ processors and $O(N^2 \log N)$ space. How is this possible? Notice that we are not sorting arbitrary real numbers; we are sorting integers between 0 and N. We rely on the following theorem:

Theorem 5 *[RR89] It is possible to sort N numbers in the range $[1, N(\log N)^{O(1)}]$ in $O(\log N)$ expected time using $O(\frac{N+M}{\log N})$ priority CRCW PRAM processors*

The algorithm of [RR89] allows us to sort the nodes by decreasing out-degree with fewer processors. However, we cannot sort the edges according to equation 1 using this method, because sorting the edges is equivalent to sorting M numbers between 0 and $N^2 - 1$. We simultaneously use an edge-array data structure and a modification of an adjacency-matrix data structure defined as follows:

$$U[i][j] = \begin{cases} k \text{ if } (u_i, u_j) \text{ is stored in position } k \text{ of EA.} \\ \text{uninitialized otherwise. May contain any value.} \end{cases}$$

Thus, only M elements of U will be initialized. Now to test in constant time whether an edge (u_i, u_j) is in G, we check whether EA[U[i][j]] $= (u_i, u_j)$ (see figure 4). Note that U is not a boolean matrix; each matrix element stores integers ranging from 0 to M. There, each matrix element can have size $\log N$; U thus uses $O(N^2 \log N)$ space.

The algorithm is as follows:

Algorithm 3 (Interval Order Recognition)
Input: A transitively-closed graph G represented in edge-array form.
Output: Returns **true** if G is an interval order and **false** if not.

{Implementation of Step 1 of Algorithm 1}
Sort the nodes by descending outdegree using the sort of [RR89].
Let $u_0, u_1, \ldots u_{N-1}$ be the nodes in topological order.
For all $0 \leq i < M$ **do on** $\frac{M}{\log N}$ **processors working in parallel**
Let (u_j, u_k) be the edge represented in array position $EA[i]$.
    U$[j][k] \leftarrow i$
{Implementation of Step 2 of Algorithm 1 }
For all $0 \leq i < M$ **do on** $\frac{M}{\log N}$ **processors working in parallel**
Let (u_j, u_k) be the edge represented in array position $EA[i]$.
    If $j > 0$ **then**
        If $0 \leq$U$[j-1][k] < M$ **then**
            If EA[U$[j-1][k]$] stores (u_{j-1}, u_k) **then**
                ReturnValue$[i]$ $\leftarrow$ **true**
            Else
                ReturnValue$[i]$ $\leftarrow$**false**
        Else
            ReturnValue$[i]$ $\leftarrow$**false**
    Else
        ReturnValue$[i]$ $\leftarrow$ **true**
Return $\bigwedge_{i=0}^{M}$ ReturnValue$[i]$

Theorem 6 *Algorithm 3 determines whether a transitively-closed graph G is an interval order. It runs on priority CRCW PRAM in $O(\log N)$ expected time using $O(\frac{N+M}{\log N})$ processors and $O(N^2)$ space.*

Proof: Algorithm 3 simulates Algorithm 1. The bound on the number of processors and the space follows directly from Theorem 5. □

3.4 A Constant-Time Algorithm

The topological sort of Step 1 of Algorithm 1 would seemingly limit the running time of a parallel algorithm to $O(\log N)$. In fact, we can create an algorithm which runs in constant time.

As in the algorithm of [PY79], we use an *adjacency-matrix* structure. Let $u_0, u_1, \ldots u_{n-1} \in V$ be the nodes of graph $G = (V, E)$, with the indices chosen arbitrarily. Then adjacency matrix U for graph G is defined as follows:

$$U_{ij} = \begin{cases} 1 & \text{if } (u_i, u_j) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Using this structure, we can test whether a given edge is in graph G in constant time. We also store the out-degrees of the nodes (see figure 5). We use the following topological order:

$$u_i < u_j \Leftrightarrow \begin{cases} \text{either} & d^+(u_i) > d^+(u_j) \\ \text{or} & d^+(u_i) = d^+(u_j) \text{ and } i < j. \end{cases} \tag{3}$$

Two subtle points allow us to create the constant-time algorithm. First, in fact, we do not need to *sort* the nodes. Instead, for each node, u_i, we only need to determine its immediate successor in a topological order. Second, we choose to design an algorithm for priority CRCW PRAM. Now, given a set of distinct integers $d_0, d_1, d_2, \ldots$, between 0 and N, we can find the minimum in constant

0	1	2	3	4	5	6	7	8	9	10	11	12	13
$(1,6)$	$(0,5)$	$(0,4)$	$(3,6)$	$(1,4)$	$(1,5)$	$(2,6)$	$(2,4)$	$(4,5)$	$(0,6)$	$(3,5)$	$(0,3)$	$(4,6)$	$(2,5)$

$$\mathbf{U} = \begin{pmatrix} - & - & - & 11 & 2 & 1 & 9 \\ - & - & - & - & 4 & 5 & 0 \\ - & - & - & - & 7 & 13 & 6 \\ - & - & - & - & - & 10 & 3 \\ - & - & - & - & - & 8 & 12 \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{pmatrix}$$

Figure 4: A coding of the graph represented in figure 2a. The uninitialized spaces of the array are indicated with a dash, and may contain any data. This data structure is used by Algorithm 3, because it allows us to determine in constant time whether an edge is in the graph without having to initialize the entire matrix. To verify that edge (u_2, u_4) is in the graph in constant time, we notice that 7 is stored in position U[2][4] and then that $(2,4)$ is stored in position 7 of the adjacency array.

time with N processors. We choose one memory location. For each i, processor p_i writes value d_i to the memory location. Because of the CRCW priority model, the value which is written to the memory location is the value written by the smallest processor, therefore the smallest d_i. (Of course, our algorithm is more complicated, since our d_i's are not all distinct.)

The algorithm maintains the following variables (see figures 6 and 7):

bool ReturnValue
int array DegExist$[0, \ldots, N-1]$
int array DegNext$[0, \ldots, N-1]$
int array NextNode$[0, \ldots, N-1]$
int array U$[0, \ldots, N-1][0, \ldots, N-1]$.

The structure DegExist is used to determine in constant time whether there exists a node with outdegree i for $0 \leq i < N$:

$$\text{DegExist}[i] = \begin{cases} 1 & \text{if } \exists u_j \text{ such that } d^+(u_j) = i\,, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

DegNext acts like an array of pointers to array elements of DegExist. We denote the NIL pointer by Λ, which can be represented in the machine by any integer not in $[0, N)$. DegNext is a structure used to determine in constant time the next smallest outdegree which exists in the graph—that is

$$\text{DegNext}[i] = \begin{cases} \max j & \text{such that DegExist}[j] = 1 \text{ and } j < i\,, \\ \Lambda & \text{otherwise.} \end{cases} \tag{5}$$

For each i such that DegExist$[i] = 0$, NextNode$[i] = \Lambda$. Otherwise, NextNode$[i]$ is j where u_j is the immediate successor of u_i in the topological sort defined in equation (3). That is, if there exists

$$\mathbf{U} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$d^+ \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 2 \\ 0 \\ 3 \\ 2 \\ 4 \end{pmatrix}$$

Figure 5: An adjacency-matrix data structure of the graph represented in figure 2a. This data structure is used by Algorithm 4 and Algorithm 6, because it allows us to determine in constant time whether an edge is in the graph. The out-degrees of the nodes are also stored.

a j such that $d^+(u_i) = d^+(u_j)$ and $i < j$ then

$$\text{NextNode}[i] = \min j \text{ such that } d^+(u_i) = d^+(u_j) \text{ and } i < j. \tag{6}$$

Otherwise,

$$\text{NextNode}[i] = \min j \text{ such that } d^+(u_j) = \text{DegNext}[d^+(u_i)]. \tag{7}$$

Matrix U is the adjacency matrix described in equation (2).

DegNext	λ	0	λ	1	3	λ	λ

DegExist	1	1	0	1	1	0	0

NextNode	3	4	0	λ	5	2	1

Figure 6: The data structure used by Algorithm 4, illustrated with the graph represented in figure 2a. Variables DegNext, DegExist, NextNode are presented as they would be calculated in Algorithm 4.

Algorithm 4 (Adjacency Matrix Implementation)
Input: A transitively-closed graph G represented in adjacency-matrix form and the outdegrees of the nodes.
Output: Variable ReturnValue is **true** if G is an interval order and **false** if not.

```
    {Implementation of Step 1 of Algorithm 1 }
                                                        {Initialize arrays ...}
1   For all 0 ≤ j < N in parallel do on proc(j)
2       DegExist[j] ← 0
3       DegNext[j] ← Λ
4       NextNode[j] ← Λ
                                                        {Construct DegExist ...}
5   For all 0 ≤ j < N in parallel do on proc(j)
6       DegExist[d+(u_j)] ← 1          {Concurrent write (CW) but same value written}
                                                        {Construct DegNext ...}
7   For all 0 ≤ i, j < N in parallel do on proc(i * N + n − 1 − j)
8       If DegExist[i] = 1 then
9           If DegExist[j] = 1 and j < i then
10              DegNext[i] ← j                    {CW. Largest j will be written}
                                                        {Construct NextNode ...}
11  For all 0 ≤ i, j < N in parallel do on proc(i * N + j)
12      If DegNext[i] = d+(u_j) then
13          NextNode[i] ← j                       {CW. Smallest j will be written}
14      Else if d+(u_i) = d+(u_j) and j > i then
15          NextNode[i] ← j                       {CW. Smallest j will be written}
    {Implementation of Step 2 of Algorithm 1 }
16  ReturnValue ← true
17  For all 0 ≤ i, j < N in parallel do on proc(i * N + j)
18      If U[i][j] = 0 and U[NextNode[i]][j] = 1 then
19              ReturnValue ← false               {CW but same value written}
```

Theorem 7 *Algorithm 4 determines whether graph G is an interval order. It runs in constant time on $O(N^2)$ processors using $O(N^2)$ space.*

Proof: It is sufficient to prove that Algorithm 4 is an implementation of Algorithm 1. Steps 1-15 of Algorithm 4 execute step 1 of Algorithm 1. Note that the maximum of equation (5) and the minima of equations (6) and (7) are computed implicitly because of the priority CRCW PRAM model: The desired extrema is always written by the processor with the smallest index. That steps 16-19 of Algorithm 4 execute step 2 of Algorithm 1 follows directly from the definition of adjacency matrices, equation (2). There are a constant number of steps to execute—hence the constant time bound. The processors used in Algorithm 4 are uniquely indexed from 0 to $N^2 - 1$—hence the $O(N^2)$ bound on the number of processors. All arrays used in the algorithm have constant-size array elements and there are $O(N^2)$ array elements—hence the $O(N^2)$ space bound. $\square$

4 Parallel Algorithms for Constructing Interval Representations

4.1 Properties of Interval Representations

In the following section, we provide algorithms which determine a corresponding set of intervals given an interval order. First, however, we must develop some machinery necessary to understand the algorithms.

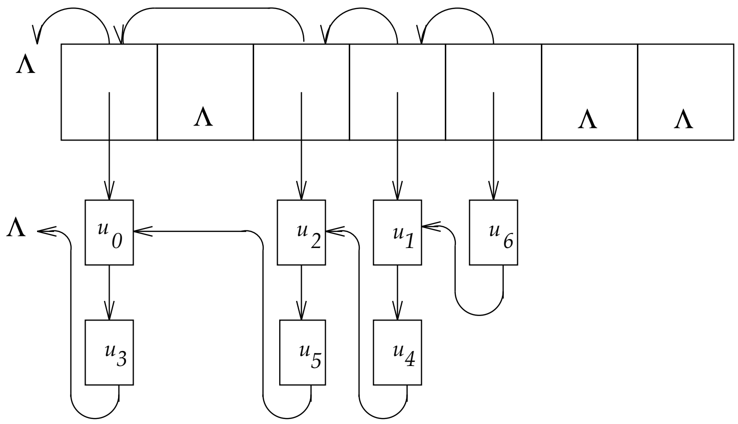

Figure 7: The data structure which variables DegNext, DegExist, NextNode of Algorithm 4 implicitly represent, illustrated with the graph from figure 2a. The topological order calculated is $u_6, u_1, u_4, u_2, u_5, u_0, u_3$.

We assume that intervals are of integer lengths, and that they are closed at the lower end of the interval and open at the upper end of the interval. The algorithms will generate interval representations with intervals lying within the range $[0, N)$.

Let $u_0, u_1, \ldots u_{n-1} \in V$ be the nodes of graph $G = (V, E)$ in arbitrary order. We represent the interval corresponding to each node by storing the endpoints of the interval. Then Upper$[u_i]$ is the open upper endpoint of interval I_{u_i} and Lower$[u_i]$ is the closed lower endpoint of interval I_{u_i}.

Proposition 3 *Let $G = (V, E)$ be an interval order. Let Upper$[0, 1, \ldots, n-1]$ be a corresponding set of integer interval upper endpoints for nodes $u_0, u_1, \ldots u_{n-1}$ of an interval order. Then,*

1. *Upper$[u_i]$ = Upper$[u_j] \Rightarrow d^+(u_i) = d^+(u_j)$.*
2. *$d^+(u_i) > d^+(u_j) \Rightarrow$ Upper$[u_i] <$ Upper$[u_j]$.*

Proposition 4 *Suppose that integer array Upper$[0, 1, \ldots, n-1]$ satisfies statements 1 and 2 of Proposition 3 for interval order $G = (V, E)$. Let*

$$Lower[u_i] = \begin{cases} \min_{0 \le j < N}(Upper[u_j]) - 1 \text{ if } Pred(u_i) = \emptyset \\ \max_{0 \le j < N}(Upper[u_j]) \text{ such that } u_j \in Pred(u_i) \text{ otherwise.} \end{cases}$$

Then, Lower and Upper form an interval representation of G with intervals I_{u_i} open at Upper$[u_i]$ and closed at Lower$[u_i]$.

Proof: It is necessary to prove that

$$\forall i, j\ [(u_i, u_j) \in E] \Leftrightarrow \text{Upper}[i] \le \text{Lower}[j].$$

($\Rightarrow$) Suppose that $(u_i, u_j) \in E$. By the definition of Lower$[j]$, Upper$[u_i] \le$ Lower$[u_j]$.
($\Leftarrow$) Suppose that $(u_i, u_j) \notin E$. If $Pred(u_j) = \emptyset$ then by the definition of Lower$[u_j]$, Lower$[u_j] <$ Upper$[u_k]$, for all k. Therefore, Upper$[u_i] \not\le$ Lower$[u_j]$. Otherwise, by the definition of Lower$[u_j]$, Lower$[u_j]$ = Upper$[u_k]$ where u_k is a predecessor of u_j with maximum Upper$[u_k]$. Since $(u_k, u_j) \in E$ and $(u_i, u_j) \notin E$ by the equivalence of statements 1 and 3 of Theorem 1, $d^+(u_i) < d^+(u_k)$. By Proposition 3, Upper$[i] >$ Upper$[k]$. Therefore, Upper$[i] \not\le$ Lower$[j]$.

It is also necessary to demonstrate that all the intervals are at least of length 1: If u_i has no successors, then by the definition of Lower, $\forall j, \text{Lower}[u_i] < \text{Upper}[u_j]$—this condition is true in particular for the case of u_i. Otherwise, consider all predecessors u_j of u_i. By Proposition 3, $\text{Upper}[u_j] < \text{Upper}[u_i]$. Therefore all intervals are at least of length 1. $\square$

We can relate the above properties to the set of maximal antichains of G. Let $\text{MA}(G)$ be the set of maximal antichains in graph G, and let $|\text{MA}(G)|$ be the size of this set.

Theorem 8 *Consider an interval representation $\mathcal{I}_G$ with intervals of integer lengths. The smallest possible interval denoted $Rng(G)$ which entirely contains $\mathcal{I}_G$ is of length greater than or equal to $|MA(G)|$. There exists an interval representation where $Rng(G)$ is of length $|MA(G)|$.*

Proof: We generate a set of antichains from an interval representation of G in the following manner: Consider the values l such that $l = \text{Lower}[u_i]$, for each u_i. For each distinct value of l we can generate an antichain so that

$$u_j \in A_l(G) \text{ if and only if } l \in I_{u_j}.$$

Clearly, all maximal antichains must be in this set of antichains. Therefore, there must be at least $|\text{MA}(G)|$ different values for l and each interval is at least of length one.

If we number the maximal antichains of G according to theorem Theorem 1, statements 5, $AM_1(G), AM_2(G), AM_3(G), \ldots$, we can immediately find an interval representation such that $Rng(G) = |\text{MA}(G)|$. We let $\text{Upper}[u_i]$ equal the index of the last antichain which contains u_i and we let $\text{Lower}[u_i]$ equal one less than the index of the first antichain which contains u_i. $\square$

Let $\text{MA}(G)$ be the set of maximal antichains in graph G. We define the measure of compactness of an arbitrary interval representation with interval endpoints is

$$\left[\max_{0 \le i < N} (\text{Upper}[i]) - \min_{0 \le i < N} (\text{Lower}[i])\right] / \, |\text{MA}(G)|.$$

Proposition 5 *Let L_G be the number of distinct out-degrees in $G = (V, E)$. Let $\mathcal{I}_G$ be an integer interval representation of G. Let $C(\mathcal{I}_G)$ be the compactness of interval representation $\mathcal{I}_G$. Then, $L_G \le C(\mathcal{I}_G)\,|MA(G)|$.*

Proof: The proof follows directly from Proposition 3. $\square$

Proposition 6 *Let Upper$[i]$ be the number of distinct out-degrees of nodes in G that are less than or equal to $d^+(u_i)$. Let Lower$[i]$ be defined according to Proposition 4. Arrays Lower and Upper form the most compact interval representation of G.*

Proof: The proof follows directly from Proposition 4 and Proposition 3. $\square$

4.2 Algorithms for Finding the Most Compact Interval Representations

Proposition 6 suggests an algorithm which finds interval representations of interval orders (see figure 8). The algorithm maintains the following variables:

int array Upper$[0, \ldots, N-1]$
int array Lower$[0, \ldots, N-1]$
int array New$[0, \ldots, N-1]$

and uses an edge-array data structure.

Array elements Upper$[i]$ and Lower$[i]$ define the open upper and closed lower endpoints of interval I_{u_i}. The array New determines whether or not u_i is the first node in the topological order with outdegree $d^+(u_i)$:

$$\text{New}[i] = \begin{cases} 0 & \text{if } d^+(u_i) = d^+(u_{i-1}), \\ 1 & \text{otherwise.} \end{cases}$$

We greatly benefit from defining New in this manner because the partial sum of New is, in fact, Upper, as it is defined in Proposition 6:

$$\text{Upper}[i] = \sum_{j=0}^{i} \text{New}[j].$$

The array Lower is defined according to Proposition 4.

Algorithm 5
Input: An interval order $G = (V, E)$ represented in edge-array or adjacency-matrix form.
Output: The most compact interval representation of G. Upper$[i]$ and Lower$[i]$ are the open upper and closed lower endpoints of I_{u_i}.
Preprocessing: Sort the nodes in V by descending out-degree. Let $u_0, u_1, \ldots u_{N-1}$ be the nodes in the corresponding topological order. Sort the edges of array EA by ascending index of the second component of each couple.

1 **For all** $0 \leq i < N-1$ **in parallel do**
2   **If** $d^+(u_{i+1}) = d^+(u_i)$ **then**
3   New$[i+1] \leftarrow 0$
4   **Else**
5   New$[i+1] \leftarrow 1$
6 New$[0] \leftarrow 1$
7 Upper$[i] \leftarrow \sum_{j=0}^{i}New[j]$
8 **For all** $0 \leq i < N$ **in parallel do**
9   Lower$[i] \leftarrow 0$
10 **For all** $0 \leq i < N$ **in parallel do**
11   Lower$[i] \leftarrow \max_{j|u_j \in Pred(u_i)}(\text{Upper}[u_j])$

Theorem 9 *Algorithm 5 determines the most compact interval representation of G. It runs on EREW PRAM in $O(\log N)$ time with $O(N+M)$ processors and $O(N+M)$ space if the [Col88] sorting procedure is used. It runs optimally on CRCW PRAM in $O(\log N)$ expected time with $O(\frac{N+M}{\log N})$ processors and $O(N^2)$ space if the [RR89] sorting procedure is used.*

Proof: Steps 1-7 explicitly calculate Upper according to Proposition 6. Steps 8-11 explicitly calculate Lower according to Proposition 4.

Note that all the operations (partial sums and maximums) except for the preprocessing can be computed optimally in $O(\log N)$ time with $O(\frac{N+M}{\log N})$ processors and $O(N+M)$ space. It is therefore only necessary to choose a sorting procedure for the preprocessing. $\square$

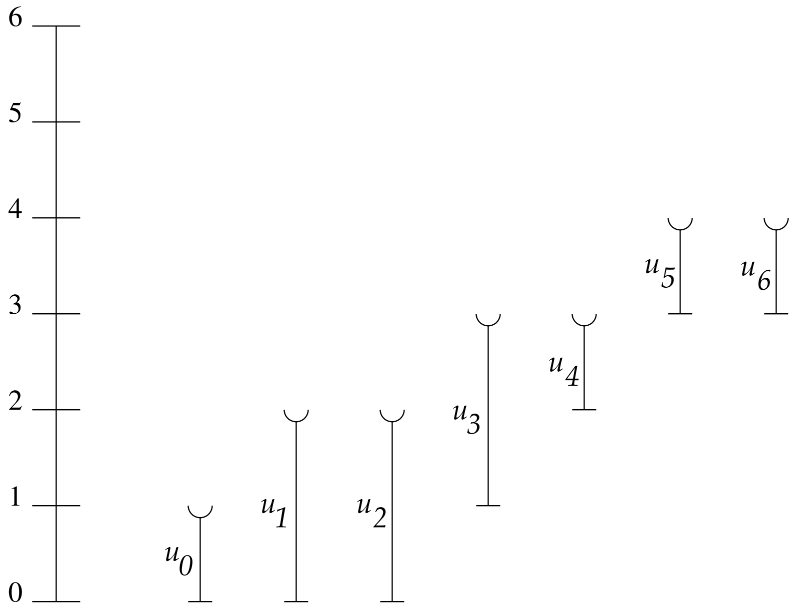

Figure 8: The compact interval representation of the graph represented in figure 2b. This representation is generated by Algorithm 5.

Corollary 1 *If, as input, we are provided the in-degrees and the out-degrees of each node, we can determine the most compact representation more efficiently. On EREW PRAM we can determine the most compact representation in time $O(\log N)$ with $O(N)$ processors if the [Col88] sorting procedure is used. On priority CRCW PRAM we can determine the most compact representation in expected time $O(\log N)$ with $O(\frac{N}{\log N})$ processors if the [RR89] sorting procedure is used.*

Proof: Steps 1-7 calculate Upper$[u_i]$ to be the number of distinct out-degrees less than or equal to $d^+(u_i)$. Since the representation is compact, we can also calculate Lower$[u_i]$ to be the number of distinct in-degrees less than $d^-(u_i)$. We need not even store and read the edges. □

4.3 A Constant-Time Algorithm

If a compact interval representation is not required, and if priority CRCW PRAM is available, then there exists an algorithm which runs in constant time (see figure 9). As before, the processors are numbered, and concurrent write conflicts are resolved by having the processor with the smallest index write its value. Our algorithm relies heavily on the following proposition:

Proposition 7 *Let $G = (V, E)$ be an interval order. There exists an interval representation of G such that Upper$[u_i] = N - d^+(u_i)$, for all i.*

Proof: The proof follows directly from Propositions 3 and 4. □

The algorithm maintains the following variables:

int array CalcMin$[0, \ldots, N-1]$
int array B$[0, \ldots, N-1][0, \ldots, N-1]$.
int array U$[0, \ldots, N-1][0, \ldots, N-1]$.
int array Upper$[0, \ldots, N-1]$
int array Lower$[0, \ldots, N-1]$

The arrays CalcMin and B are defined so that we can take the maximum and minimum of Proposition 4 in constant time using the priority CRCW PRAM model. The strategy is always the same: Assign the right task to the right processor. We want the smallest value to be written by the smallest processor when a minimum is desired and the largest value to be written by the smallest processor when a maximum is desired. The array CalcMin is defined as follows:

$$\text{CalcMin}[j] = \begin{cases} 1 & \text{if } \exists i \text{ such that Upper}[i] = j \\ 0 & \text{otherwise.} \end{cases}$$

The matrix B is defined as follows:

$$\text{B}[i][j] = \begin{cases} 1 & \text{if } \exists k \text{ such that } u_k \in \mathit{Pred}(u_i) \textit{ and } j = \mathit{Upper}[k] \\ 0 & \text{otherwise.} \end{cases}$$

Matrix U is the adjacency matrix described in equation (2). Arrays Upper and Lower are defined according to Proposition 7 and Proposition 4 respectively.

Algorithm 6
Input: An interval order $G = (V, E)$ represented in adjacency-matrix form. Nodes $u_0, u_1, \ldots u_{n-1} \in V$ are not necessarily topologically sorted.
Output: An interval representation of G. Upper$[i]$ and Lower$[i]$ are the open upper and closed lower endpoints of I_{u_i}.

1 **For all** $0 \leq i < N$ **in parallel do on** proc(i)
2   Upper$[i] \leftarrow N - d^+(u_i)$ {Assigned according to Proposition 7.}
3   CalcMin$[i] \leftarrow 0$
4 **For all** $0 \leq i < N$ **in parallel do on** proc(i)
5   CalcMin[Upper$[i]$] $\leftarrow 1$ {CW but same value will be written.}
6 **For all** $0 \leq i < N$ **in parallel do on** proc(i)
7   **If** CalcMin$[i] = 1$ **then**
8   Min $\leftarrow i$ {CW. Smallest value of i will be written.}
9 **For all** $0 \leq i < N$ **in parallel do on** proc(i)
10   Lower$[i] \leftarrow$ Min-1
11 **For all** $0 \leq i, j < N$ **in parallel do on** proc$(i * N + j)$
12   B$[i][j] \leftarrow 0$
13 **For all** $0 \leq i, k < N$ **in parallel do on** proc$(i * N + k)$
14   **If** $U[k][i] = 1$ **then**
15   B$[i]$[Upper$[i]$] $\leftarrow 1$ {CW but same value will be written.}
16 **For all** $0 \leq i, j < N$ **in parallel do on** proc$(i * N + N - j - 1)$
17   **If** B$[i][j] = 1$ **then**
18   Lower$[i] \leftarrow j$ {CW. Largest value of j will be written.}

Theorem 10 *Algorithm 6 determines an interval representation of G. It runs in constant time on $O(N^2)$ processors using $O(N^2)$ space.*

Proof: Note that the maximum and minimum of Proposition 7 is computed implicitly in steps 8 and 18 because of the priority CRCW PRAM model: The desired extrema is always written by the processor with the smallest index. There are a constant number of steps to execute—hence the constant time bound. The processors used in Algorithm 6 are uniquely indexed from 0 to N^2-1—hence the $O(N^2)$ bound on the number of processors. All arrays used in the algorithm have constant-size array elements and there are $O(N^2)$ array elements—hence the $O(N^2)$ space bound. □

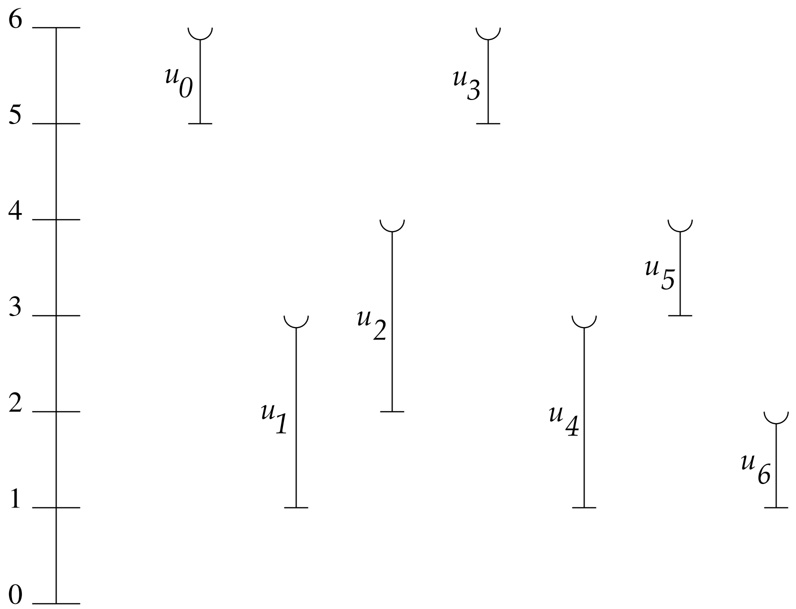

Figure 9: A non-compact interval representation of the graph represented in figure 2a. This representation is generated by Algorithm 6.

5 Algorithms for Distributed-Memory Machines

Notice that Algorithms 2 and 5 are composed of several sorting and partial sum routines. Because sorting algorithms and partial sum algorithms have been studied intensively on many distributed-memory architectures, it is a relatively small step to write a distributed-memory algorithms for recognizing and representing interval orders.

Consider a hypercube architecture. The sorting procedure for hypercube with the best complexity, proposed by Cypher and Plaxton [CP90], runs in time $O(\log N\,(\log\log N)^2)$ on $O(N)$ processors. The partial sum algorithm for hypercube introduced by Nassimi and Sahni [NS81] runs optimally in time $O(\log N)$. Therefore, on a hypercube we can implement Algorithms 2 and 5 to run in time $O(\log N\,(\log\log N)^2)$ on $O(N)$ processors.

On a $O(\sqrt{N})$ by $O(\sqrt{N})$ mesh of processors, sorting procedures with the best complexity–for example, the rotate sort of Marberg and Gafni [MG86] or the bitonic sort of Batcher [Bat68]–run optimally in time $O(\sqrt{N})$. A partial sum also runs optimally in $O(\sqrt{N})$. Therefore on a $O(\sqrt{N})$ by $O(\sqrt{N})$ mesh of processors we can implement Algorithms 2 and 5 optimally in time $O(\sqrt{N})$.

6 Future Research

In this paper we have proposed PRAM and distributed-memory algorithms for recognizing and representing interval orders. This work strongly motivates work on several open problems. Noting that our algorithms only accept transitively-closed directed acyclic graphs as input, we will try to develop a parallel algorithm which accepts a non-transitively closed directed acyclic graphs as input, and which does not require the transitive closure. Such an algorithm could permit a more efficient representation of orders. Another open problem is the following: Can one apply the ideas introduced in this paper to construct algorithms on other classes of orders? Finally, and more generally, can one determine which classical algorithms on orders are easily parallelizable?

References

[Bat68] K.E Batcher. Sorting Networks and their Applications. In *Spring Joint Computer Conf.*, pages 307– 314, 1968.

[BM93] Philippe Baldy and Michel Morvan. A linear time and space algorithm to recognize interval orders. *Discrete Applied Mathematics.*, 1993.

[Bog82] K. P. Bogart. Some social science applications of ordered sets. In I. Rival, editor, *Ordered Sets*, pages 759–787. Reidel, Dordrecht, (1982.

[Col88] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4), August 1988.

[CP90] R. Cypher and C. G Plaxton. Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers. In *22nd Annual Symposium on Theory of Computing*, pages 193–203, October 1990.

[Fis70] P. C. Fishburn. Intransitive indifference in preference theory: A survey. *Oper. Res.*, 18:207–228, 1970.

[Fis85a] P. C. Fishburn. Interval graphs and interval orders. *Discrete Math.*, 55:135–150, 1985.

[Fis85b] P. C. Fishburn. *Interval orders and Interval graphs.* Wiley, New York, 1985.

[Gab81] H. Gabow. A linear time recognition algorithm for interval dags. *Information Processing Letters*, 12(1):20–22, February 1981.

[Gar92] R. Garbe. A linear time algorithm for recognizing interval orders. Memorandum No 1046, Dept. of Math. University of Twente, March 1992.

[Gol85] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1985.

[HJ85] M. Habib and R. Jegou. N-free poses as generalizations of series-parallel posets. *Discrete Appl. Math.*, 12(3):279–291, 1985.

[MG86] J. M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. In *Proceedings of the Allerton Conference on Computing, Communication and Control*, 1986.

[Moh89] Rolf H. Mohring. Computationally tractable classes of ordered sets. In I. Rival, editor, *Algorithms and Order*, pages 105–193. Kluwer Acad. Publ., Dordrecht, 1989.

[NS81] D. Nassimi and S. Shani. Data broadcasting in SIMD computers. *IEEE Trans on Computers*, 30(2):101–107, 1981.

[PLE71] A. Pneuili, A Lempel, and W. Even. Transitive orientation of graphs and identification of permutation graphs. *Canad. J. math*, 23:160–175, 1971.

[PY79] C. H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Comp.*, 8:405–409, 1979.

[RR89] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18:594–607, 1989.

[Spi85] J. Spinrad. On comparability and permutation graphs. *SIAM J. Comput.*, 14:658–670, 1985.

[SV83] J. Spinrad and J. Valdes. Recognition and isomorphism of two dimensional partial orders. In *10th Coll. on Automata, Language and Programming.* Lecture Notes in Computer Science , volume 154, pages 676–686. Springer, Berlin, 1983.

[Sys82] M. M. Syslo. A labeling algorithm to recognize a line digraph and output its root graph. *Inform. Processing Letters*, 15:241–260, 1982.

[VTL82] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series-parallel digraphs. *SIAM J. Comput.*, 11:298–314, 1982.

[Wei14] N. Weiner. A contribution to the theory of relative position. *Proc. Camb. philos. Soc.*, 17:441–449, 1914.