ELSEVIER

Theoretical Computer Science 292 (2003) 639–652

Theoretical Computer Science

www.elsevier.com/locate/tcs

An efficient deterministic parallel algorithm for two processors precedence constraint scheduling ☆

Hermann Jung[a, 1], Maria Serna[b], Paul Spirakis[c, *]

[a] *Humboldt University, Germany*

[b] *Dept. de Llentguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain*

[c] *Computer Technology Institute, Riga Fereou 61, 26110 Patras, Greece*

Received 17 March 1995; received in revised form 22 November 2000; accepted 16 January 2001
Communicated by J. Díaz

Abstract

We present here a new deterministic parallel algorithm for the two-processor scheduling problem. The algorithm uses only $O(n^3)$ processors and takes $O(\log^2 n)$ time on a CREW PRAM. In order to prove the above bounds we show how to compute in NC the lexicographically first matching for a special kind of convex bipartite graphs.

Keywords: Scheduling; Parallel algorithms; PRAM

1. Introduction

A classical problem in scheduling theory is to find an optimal nonpreemptive schedule for a collection of unit length tasks subject to precedence constraints. We are given n tasks to be executed on m processors. Each task requires exactly one unit of execution time and can run on any processor. A directed acyclic graph specifies the precedence constraints where an edge from task x to task y means task x must be completed before task y begins. A solution to the problem is a schedule of shortest length indicating

☆ A preliminary version of our paper, titled "A parallel algorithm for the two Processors Precedence Constraint Scheduling" appeared in the 18th ICALP (1991), Springer-Verlag, Vol. 510, pp. 417–428. This work was partially supported by the ESPRIT II BRA No. 3075 (project ALCOM II) and by the IST FET Program ALCOM-FT.

* Corresponding author.

E-mail addresses: mjserna@lsi.upc.es (M. Serna), spirakis@cti.gr (P. Spirakis).

[1] Formerly at Humboldt University, Germany.

PII: S0304-3975(01)00120-7

when each task is started. When the number of processors m is arbitrary the problem is NP-complete [8]. For any $m \geqslant 3$, the complexity is open [6]. Here we study the case $m = 2$. For two processors a number of efficient algorithms has been given. For sequential algorithms see [4, 2, 5] among others. The first deterministic parallel algorithm was given by Helmbold and Mayr [7], thus establishing membership in the class NC. Previously [9] gave a randomized NC algorithm for the problem.

We present here a new parallel algorithm for the two-processor scheduling problem that takes time $O(\log^2 n)$ and uses $O(n^3)$ processors on a CREW PRAM. Our algorithm improves the number of processors of the algorithm given in [7] from $O(n^7 L(G)^2)$, where $L(G)$ is the number of levels in the precedence graph, to $O(n^3)$. Both algorithms compute a level schedule that has a lexicographically first jump sequence (see definitions below).

The Helmbold and Mayr algorithm is based on a technique to compute the schedule length in parallel. By running the above in parallel many times over different graphs, they find which levels in a given graph end with a task that is scheduled together with an empty slot, we call them jumps to level 0. By computing those levels they are able to compute a lexicographically first jump sequence. Then the last step is just to find a way to assign a task to each jump (when possible).

Our algorithm behaves quite differently. We also use an algorithm to compute schedule length, but at the same time we record a graph decomposition that allows us to compute jumps to level 0 as well. We run the length algorithm (on an adequate graph) only once, so that all the information needed in the recursive phase of the algorithm could be built in. In the recursive phase we divide the graph in three parts, tasks in level i, the central part, tasks in higher levels, the lower part, and tasks in lower levels, the upper part. Then our problem is to show how to match jumps from levels in the lower part with tasks in the upper part. Note that our algorithm never computes the whole jump sequence, only those jumps that are needed in each division. Then an algorithm to match such jumps, together with an adequate graph decomposition, gives the desired result.

To match jumps with tasks, we consider the problem of computing the lexicographically first matching for a special type of convex bipartite graphs, (called here as full convex bipartite graphs). A geometric interpretation of this problem leads to the discovery of an efficient parallel algorithm to solve it.

2. Definitions and remarks

The two-processor scheduling problem is defined by a directed acyclic graph (dag) $G = (V, E)$. The vertices of the graph represent unit time tasks, and the edges specify precedence constraints among the tasks. If there is an edge from node x to node y then x is an *immediate predecessor* of y. *Predecessor* is the transitive closure of the relation immediate predecessor, and *successor* is its symmetric counterpart. A *two-processor schedule* is an assignment of the tasks to time units $1, \ldots, t$ so that each task

is assigned exactly one time unit, at most two tasks are assigned to the same time unit, and if x is a predecessor of y then x is assigned to a lower time unit than y. The length of the schedule is t. A schedule having minimum length is an *optimal* schedule. We will assume that tasks are partitioned into levels as follows:

(i) every task will be assigned to only one level,
(ii) tasks having no successors will be assigned to level 1 and
(iii) for each level i, all tasks which are immediate predecessors of tasks in level i will be assigned to level $i+1$.

Clearly topological sort will accomplish the above partition, and this can be done by an NC algorithm that uses $O(n^3)$ processors and $O(\log n)$ time, see [3]. Thus, from now on, we will assume that a level partition is given as part of the input. For the sake of convenience we add two special tasks, t_0 and t^*, so that the original graph could be taught as the graph formed by all tasks that are successors of t_0 and predecessors of t^*. Thus, t_0 is a predecessor of all tasks in the system (actually an immediate predecessor of tasks in level the highest level $L(G)$) and t^* is a successor of all tasks in the system (an immediate successor of level 1 tasks).

Notice that if two tasks are at the same level they can be paired. But when x and y are at different levels, they can be paired only when neither of them is a predecessor of the other. Let $L(G)$ denote the number of levels in a given precedence graph G. A *level schedule* schedules tasks level by level. More precisely, suppose levels $L(G),\ldots,i+1$ have already been scheduled and there are k unscheduled tasks remaining on level i. If k is even, we pair the tasks with each other. If k is odd we pair $k-1$ of the tasks with each other and the remaining task may (but not necessarily) be paired with a task from a lower level. Given a level schedule we say that level i *jumps to level* i' ($i'<i$) if the last timestep containing a task from level i also contains a task from level i'. If the last task from level i is scheduled with an empty slot, we say that level i *jumps to level* 0. The *jump sequence* of a level schedule is the list of levels jumped to. A *lexicographically first jump schedule* is a level schedule whose jump sequence is lexicographically greater than any other jump sequence resulting from a level schedule. The following result is well known (it is due to Gabow ([5]), see also [7]).

Theorem 1 (Gabow [5]). *Every lexicographically first jump schedule is optimal.*

Given a graph G a *level partition* of G is a partition of the nodes in G into two sets in such a way that levels $0,\ldots,k$ are contained in one set (the upper part) denoted by U, and levels $k+1,\ldots,L$ in the other (the lower part) denoted by L.

Given a graph G and a level i, the *i-partition* of G (or the partition at level i) is formed by the graphs U_i and L_i defined as U_i contains all nodes x such that $\text{level}(x)<i$ and L_i contains all nodes x with $\text{level}(x)>i$. Note that each i-partition determines two different level partitions depending on whether level i nodes are assigned to the upper or the lower part.

We say that a task $x\in U_i$ is *free* with respect to a partition at level i if x has no predecessors in L_i.

3. Computing the schedule length

In this section we construct an algorithm to compute the optimal schedule length together with an adequate task decomposition. We compute the number of steps that must intervene between any two tasks and thus determine the number of time steps that must intervene between t_0 and t^*. A schedule with precisely this length is clearly a shortest schedule. We define the *schedule distance* between tasks x and x', $d(x,x')$ as the minimum number of timesteps required to schedule all tasks that are both successors of x and predecessors of x' when x is a predecessor of x', otherwise undefined. We will denote by $G(x,x')$ the subgraph of the precedence graph containing all tasks that are both successors of x and predecessors of x'.

In [2] it is shown how to construct sets of tasks $X_0, X_1, \ldots, X_k$, for any precedence graph, such that those tasks in any X_{i+1} are predecessors of all tasks in X_i, and the length of an optimal schedule equals $\sum_i \lceil |X_i|/2 \rceil$. Our algorithm will compute for any pair of tasks x and x' such that x is a predecessor of x' a set, $S(x,x')$, that contains one of such decompositions for the precedence graph $G(x,x')$.

Our algorithm *Length* below is similar to the corresponding algorithm of [7] as far as the length computation of an optimal schedule is concerned. However, it also computes more information, which is exactly the set $S(x,x')$ for any pair of tasks x and x'.

In the algorithm *Length*, let $L(A,B,\ldots)$ be the *list* of the concatenation of the lists $A,B,\ldots$.

Algorithm Length

$d_0(*,*) := 0$
$S_0(*,*) := \emptyset$
for $i := 1$ **to** $\lceil \log n \rceil$ **do**
  for all x, x' with $x < x'$ **do in parallel** (loop 1)
    for all z such that $x < z < x'$ **do in parallel** (loop 2)
      $S(z) = \{y \mid d_{i-1}(x,y) \geqslant d_{i-1}(x,z)$ and $d_{i-1}(y,x') \geqslant d_{i-1}(z,x')\}$
    end of loop 2
    $d_i(x,x') := \max_z \{d_{i-1}(x,z) + d_{i-1}(z,x') + \lceil \frac{|S(z)|}{2} \rceil\}$
      Note: The above line produces a specific z_0 which maximizes the expression to be assigned to $d_i(x,x')$.
    $S_i(x,x') := L(S(z_0))$
      Note: This list has one element, the set $S(z_0)$.
    Let $B(z_0,x')$ be the set of all tasks y such that (1) $y \notin S(Z_0)$ (2) $y \neq x'$ and (3) y is in between (in the predecessor relation) any of the tasks in $S(z_0)$ and x'.
    If there are t,t' such that t is a predecessor of all tasks in $S(z_0)$ and t' is a predecessor of all tasks in $B(z_0,x')$,
      then $S_i(x,x') = L(S_{i-1}(x,t), S(z_0), S_{i-1}(t',x'))$
  end of loop 1

$d(*,*) := d_{\lceil \log n \rceil}(*,*)$
$S(*,*) := S_{\lceil \log n \rceil}(*,*)$
end of algorithm Length

Algorithm length has a straightforward implementation on a CREW PRAM using $O(n^3)$ processors and $O(\log^2 n)$ time.

By following the schedule distance proof of Lemma 3 of [7] exactly, and by the way $S_i(x,x')$ is composed as a list of sets we have

Lemma 2. *Algorithm Length correctly computes* $d(x,x')$ *for all tasks* x *and* x' *such that* x *is a predecessor of* x'. *Furthermore,* $\sum_{X \in S(x,x')} \lceil |X|/2 \rceil = d(x,x')$.

Thus the length of an optimal schedule for G is $d(t_0,t^*)$. Our second result relies on the task decomposition obtained for a graph $G(x,x')$. Our aim is to use this decomposition to compute which levels jump to level 0. To do this we have only to consider those sets that have an odd number of tasks. In such a case the lower level in the set will jump to level 0 if and only if there are no tasks outside the decomposition that can be scheduled together. In order to preserve lexicographically first jump sequences, we have only to match (when possible) a level with a task in the highest possible level. Thus we have

Lemma 3. *There is a deterministic parallel algorithm to compute those levels that jump to level* 0 *for a given precedence graph, that uses* $O(n^3)$ *processors and* $O(\log^2 n)$ *time.*

4. The matching problem

A *full convex bipartite graph* G is a triple (V,W,E), where $V = \{v_1,\ldots,v_k\}$ and $W = \{w_1,\ldots,w_{k'}\}$ are disjoint sets of vertices. Furthermore, the edge set E satisfies the following property: if $(v_i,w_j) \in E$ then $(v_q,w_j) \in E$ for all $q \geqslant i$. We also assume that the graph is connected. Fig. 1 gives an example of a full convex bipartite graph.

A set $F \subseteq E$ is a *matching* in the graph $G = (V,W,E)$ iff no two edges in F have a common endpoint. We want to compute the lexicographically first maximal matching in G. This matching can be computed by the following sequential algorithm:

Lexicographically first matching
$M := \emptyset$
for $i := 1$ to k **do**
  Find the first node w in W with $(v_i,w) \in E$
  if w exists **then**
    $M := M \cup \{(v_i,w)\}$
    $W := W - \{w\}$

Fig. 2 gives the lexicographically first matching for the graph of Fig. 1.

v_9	x	x	x	x	x	x	x
v_8	x	x	x	x	x	x	x
v_7	x	x	x		x	x	x
v_6	x	x	x		x	x	x
v_5	x		x		x	x	
v_4	x		x		x	x	
v_3			x		x	x	
v_2			x		x	x	
v_1			x				
	w_1	w_2	w_3	w_4	w_5	w_6	w_7

Fig. 1. A full convex bipartite graph.

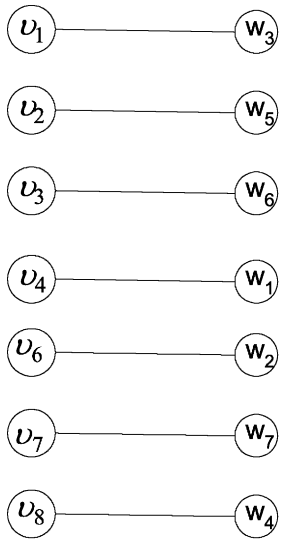

Fig. 2. The lexicographically first matching.

Before describing the parallel algorithm let us visualize in the plane our matching problem. Each index i of the sequence $(v_1,\ldots,v_k)$ of nodes in V corresponds to the y-coordinate i, while each index j of a node in the sorted sequence $(w_1,\ldots,w_{k'})$ corresponds to the x-coordinate j. Furthermore, we have k' intervals, one for each node in W. Every interval is parallel to the y-axis and has one endpoint on the k-value, the

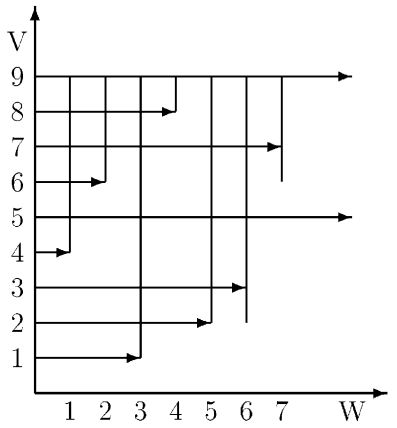

Fig. 3. Geometric representation of the matching problem.

second endpoint for the interval corresponding to w_j is defined by the minimal index i of a node $v_i \in V$ such that $(v_i, w_j) \in E$. Our matching problem can now be formulated as follows:

> Starting with 1, remove subsequently the first interval which can be seen on the horizontal ray emanating from i, one per each y-coordinate (if possible). The removed intervals form the sequence of nodes in W matched with the nodes in V.

In Fig. 3 we can see the graphic representation corresponding to the graph in Fig. 1, arrows point to the matched intervals or to infinity, if there is no interval to be matched with.

To compute the lexicographically first matching, we start by defining two matrices A and B. Let a_{ij} $(1 \leqslant i \leqslant k, 1 \leqslant j \leqslant k')$ be the number of nodes $v_{i'}$ with $i' < i$ which are matched with nodes $w_{j'}$ with $j' > j$ or which cannot be matched, and let b_{ij} be the number of intervals $w_{j'}$ $(j' \leqslant j)$, hit, by the horizontal ray i. We first show that all that we need to do to solve our matching problem is to compute matrices A (i.e. all a_{ij}) and B (i.e. all b_{ij}) both matrices. In Figs. 4 and 5 the values of matrices A and B are given for the graph in Fig. 1.

Note that $a_{ij} + b_{ij} \geqslant i - 1$ for all i, j values. To simplify notation we say that v_i hits w_j when $a_{ij} + b_{ij} = i$ and j is the smallest over all indices k such that $a_{ik} + b_{ik} = i$. An easy counting argument taking into account the definition of matrices A and B shows

Lemma 4. *Node v_i matches*

before j if and only if $a_{ij} + b_{ij} > i$, or $a_{ij} + b_{ij} = i$ and v_i does not hit w_j,

with j if and only if $a_{ij} + b_{ij} = i$, and v_i hits w_j,

behind j if and only if $a_{ij} + b_{ij} = i - 1$.

Once we have computed matrices A and B, the corresponding matched pairs can be computed as follows: first note that for a fixed i we only need the row i of matrices

9	7	6	5	4	3	2	1
8	6	5	4	4	3	2	1
7	5	4	3	3	2	1	1
6	4	4	3	3	2	1	1
5	3	3	2	2	1	0	0
4	3	3	2	2	1	0	0
3	2	2	1	1	0	0	0
2	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
i/j	1	2	3	4	5	6	7

Fig. 4. Matrix A.

9	1	2	3	4	5	6	7
8	1	2	3	4	5	6	7
7	1	2	3	3	4	5	6
6	1	2	3	3	4	5	6
5	1	1	2	2	3	4	4
4	1	1	2	2	3	4	4
3	0	0	1	1	2	3	3
2	0	0	1	1	2	3	3
1	0	0	1	1	1	1	1
i/j	1	2	3	4	5	6	7

Fig. 5. Matrix B.

a_{ij}, b_{ij}. Thus we can do the computation independently for each i. Let us fix i and consider a matrix D_i (Fig. 6) such that $D_i[j]=1$ when $a_{ij}=i-b_{ij}$ and 0 otherwise. Compute prefix sums of D in a matrix C_i (Fig. 7) (now $C_i[k]=D_i[1]+\cdots+D_i[k]$), and there is a unique j such that $D_i[j]=C_i[j]=1$ (the one we look for).

The matching is indicated in heavy squares where $D_i[j]=C_i[j]=1$.

9	0	0	0	0	0	0	0
8	0	0	0	1	1	1	1
7	0	0	0	0	0	0	1
6	0	1	1	1	1	1	0
5	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1
3	0	0	0	0	0	1	1
2	0	0	0	0	1	0	0
1	0	0	1	1	1	1	1
i/j	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Fig. 6. Matrix $D_i[j]$.

9	0	0	0	0	0	0	0
8	0	0	0	1	2	3	4
7	0	0	0	0	0	0	1
6	0	1	2	3	4	5	5
5	0	0	0	0	0	0	0
4	1	2	3	4	5	6	7
3	0	0	0	0	0	1	2
2	0	0	0	0	1	1	1
1	0	0	1	2	3	4	5
i/j	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Fig. 7. Matrix $C_i[j]$.

As prefix sums can be computed in time $O(\log n)$ using $O(n/\log n)$ processors, and we can find the corresponding j for all values of i in parallel, we can compute the matching in time $O(|w|)$ using $O(|V||w|/\log|w|\})$ processors.

Since $b_{i,j}$ can be easily computed (for $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant k'$) in parallel time $O(\log(|V|+|W|))$ using $O(|V||W|)$ processors, the matching problem is then reduced to the problem of computing matrix A. Furthermore looking at the columns of matrix

A, $a_{i+1,j}$ will be $a_{ij}+1$ only when node v_i matches behind j. Thus from Lemma 4 we get

Lemma 5. *Matrix A can be computed independently for each column j applying the following rule*: $a_{1,j}=0$ *for* $j=1,\ldots,k'$ *and*

$$a_{i+1,j}=\begin{cases} a_{i,j} & if\ \ a_{i,j}+b_{i,j}\geqslant i,\\ a_{i,j}+1 & if\ \ a_{i,j}+b_{i,j}<i.\end{cases}$$

In other words, there are thresholds $c_{i,j}$ for each (i,j) such that: $a_{i+1,j}=a_{i,j}$ when $a_{i,j}\geqslant c_{i,j}$ or $a_{i+1,j}=a_{i,j}+1$ otherwise. Furthermore $c_{i,j}=i-b_{i,j}$.

In order to compute the matrix A we associate an array $A_{i,j}[0\cdots k-1]$ of pointers to each pair (i,j). $A_{i,j}[x]$ points to $A_{i+1,j}[x]$ if $x\geqslant c_{i,j}$, and it points to $A_{i+1,j}[x+1]$ if $x<c_{i,j}$. Only $A_{i,j}[k-1]$ points in any case to $A_{i+1,j}[k-1]$. Thus each column of matrix A consists of a $k\times k$ pointer matrix that can be thought of being a set of disjoint trees with roots in the last row of the matrix. If $A_{k,j}[x]$ is the root of the tree with leaf element $A_{1,j}[0]$ then x is the correct value of $a_{k,j}$. Moreover, the indices of the nodes on the path from $A_{1,j}[0]$ to $A_{k,j}[x]$ are the values of the corresponding variables $a_{i,j}$ $(1\leqslant i\leqslant k)$ (that is a consequence of the above observation about the matrix $A=(a_{i,j})$).

It remains to compute the chain of indices on such a path. This can be done by pointer doubling, simultaneously over the whole $k\times k$ pointer matrix. This yields an $O(|V|^2|W|/\log|V|)$ processors implementation on a CREW PRAM computing all values $a_{i,j}$. Hence, we can conclude:

Lemma 6. *The lexicographically first matching of full convex bipartite graphs can be computed in time* $O(\log n)$ *on a CREW PRAM with* $O(n^3/\log n)$ *processors, where n is the number of nodes.*

5. An outline of the algorithm

Before describing the algorithm let us study the jump's structure for a partition at level i. Consider the decomposition into L_i and U_i. We consider only the possible jumps between levels in the different parts, we have three cases:

(1) Some levels in L_i should be paired (when possible) with tasks in U_i,
(2) Some levels in L_i may have to be paired with tasks in level i and
(3) Level i may have to be paired with a task in U_i.

Clearly, these levels can be obtained by computing the jumps to level 0 for the graphs L_i and $L_i\cup\{\text{nodes in level } i\}$. All these levels have to be paired (when possible) with tasks that have no predecessor in L_i, except for the last level in L_i or level i. Our algorithm starts by finding a "central" level, that is a level such that the corresponding lower and upper part have at most half of the size of the original graph. Then we solve the above mentioned jumps from the lower part levels. To keep all the information

needed we start by a preprocessing step to keep all the information needed in the remaining steps of the algorithm. We consider the following algorithm (see details in Sections 6 and 7):

Algorithm Schedule
0. Preprocessing.
1. Find a level i such that $|U_i| \leqslant n/2$ and $|L_i| \leqslant n/2$.
2. Match levels that jump to free tasks in level i.
3. Match levels that jump to free tasks in U_i.
4. If level i (or $i+1$) remain unmatched try to match it with a nonfree task.
5. Delete all tasks used to match jumps.
6. Apply (1)–(5) in parallel to L_i and the modified U_i.

Algorithm Schedule stops whenever the corresponding graph has only one level.

6. The preprocessing step

The first step in algorithm Schedule is intended to compute all information needed in the remaining steps. Our aim is to run only once the algorithm that computes the schedule length in an adequate precedence graph obtained from the original precedence graph. Let us analyze the requirements of the algorithm Schedule.

We have to compute which levels in a lower part (for all possible lower parts) jump to levels in the upper part. Notice that after splitting the graph a "lower" part L neither corresponds exactly to levels $L(G),\ldots,i$ for some i, nor contains all nodes in the corresponding levels. However, when i is the lowest level in L the jumps to level 0 of levels included in L are still the jumps to level 0 for this part in the lexicographically first schedule. Thus we have to compute jumps to level 0 in the lower part of a partition at level i for all possible values of i. To do this we modify the precedence graph G by adding a node t_i for $i=1,\ldots,L(G)$. Node t_i will be a successor of all nodes in level i, thus we add an edge (x,t_i) for all x in level i. Clearly the jumps to level 0 (for each partition) can be obtained from the graph decomposition corresponding to the nodes $G(t_0,t_i)$.

In step (4) we need to know whether a task in level $i-1$ can be scheduled together with a task in level i (for some values of i). We can compute this information for all levels modifying the graph as follows: We add a task t_x for each task x in G, t_x will be a successor of x and a successor of all tasks in level $level(x)+1$. Suppose that $level(x)=i-1$. Now whenever the schedule length of the graph $G(t_0,t_i)$ equals the schedule length of the graph $G(t_0,t_x)$ task x can be scheduled together with level i. Thus we can conclude

Theorem 7. *Given a precedence graph G, we can compute all levels that jump to level 0 in the graph L_i and all tasks in level $i-1$ that can be scheduled together with a*

task in level i, for $i = 1, \ldots, L(G)$, *in parallel using* $O(n^3)$ *processors and* $O(\log^2 n)$ *time.*

7. Computing the schedule

Let us show now how to compute the remaining steps of algorithm Schedule. In steps (2) and (3) we have to match (when possible) levels that jump to level 0 with free tasks, sometimes we have to choose between different tasks at the same level. Thus we need to define some task order, inside a level, that preserves lexicographically first jump sequences. We assign a number to each task at a given level i. Let $l(x)$ be the length of the longest path ending at x. The correctness of the use of l-values to solve ties follows from a result of [7]:

Lemma 8 (Helmbold and Mayr [7]). *Let* $l_1, l_2, \ldots, l_r$ *be the levels jumping to level* l *in descending order. Assume we have a collection of tasks* $x_1, \ldots, x_r$ *from level* l, *and each* x_i *has* $l(x_i) \geqslant l_i$. *Then there is an lexicographically first schedule such that* x_i *is used for the jump from* l_i.

Thus in step (2) we only have to sort the set of free tasks in increasing order of l-values and assign jumps to tasks in order.

To solve the assignment problem of step (3) we give a reduction to the problem of computing a lexicographically first matching for a full convex bipartite graph. The reduction is the following:

In our application we are given a level partition together with a number of levels in which one task remains to be matched with some other task in the upper part of the graph (say levels $l_1 > l_2 > \cdots > l_k$). To each of the levels in the lower part that has to be matched ($l_1 > l_2 > \cdots > l_k$) we assign a vertex v ($v_1 > v_2 > \cdots > v_k$). Furthermore we consider all free tasks in the upper part and assign a vertex to each one. These tasks are the candidates for matching the vertices in the lower part. Thus we have two sets of vertices: one for levels in the lower part and one for free tasks in the upper part.

We assign a new number to each vertex. Let V denote the set of vertices coming from the lower part, and let W denote the set of vertices assigned to tasks in the upper part. For all $v \in V$ let $h(v) = L(G) - level(v)$. For all $w \in W$ let $h(w) = L(G) - level(w)$. Now we sort all vertices in V according to increasing h-values, and all nodes in W according to increasing h-values and increasing l-values (for nodes with the same h-value).

In order to assign to each level the corresponding task (if any) we have to compute the lexicographically first matching in the bipartite graph $H = (V, W, E)$ where $(v, w) \in E$ if and only if $l(w) \leqslant l(v) = h(v) \leqslant h(w)$. Note that $h(v)$ is less than $h(w)$ for all v, w in the graph thus H is a full convex bipartite graph.

Theorem 9. *Given a level partition of a graph G together with the levels in the lower part in which one task remains to be matched with some other task in the upper part of the graph. We can compute the corresponding tasks in time* $O(\log n)$ *using* $n^3/\log n$ *processors.*

Finally step (4) can be solved using the information built in the preprocessing step. Simply the algorithm finds a task that can be paired together with such a level (note that the corresponding task must be nonfree). The correctness of the algorithm after the removal of such a task is obtained from the following. First, the jump structure is computed at the beginning of the algorithm and second, such a task can only be in a lower part in the decomposition created by algorithm Schedule.

Thus at the end of algorithm Schedule, we have a task partition and the only thing that we need is a scheduling policy. The first time step assigned to each partition can be computed by using list ranking. Once we know this timestep, each partition can be scheduled independently.

For those sets in which a jump is matched, we have to choose the task to be paired with the task from the other level, whatever task can be used when the jumped task was a free task. When the jumped task was nonfree, the jump is only possible if (at least) one of the other tasks is not a predecessor of the jumped task. Once we have this pair of task we assign to them the last timestep. The rest of the tasks will be assigned as follows: sort all tasks in increasing order of degree and assign in order pairs to timesteps. Clearly this policy can be implemented in parallel, using fewer than $O(n^3)$ processors, thus we have

Theorem 10. *There is an NC algorithm which finds an optimal two processors schedule for any precedence graph in time* $O(\log^2 n)$ *using* $O(n^3)$ *processors.*

8. Conclusions and open problems

We have presented a new parallel deterministic algorithm for the two processors scheduling problem. Our algorithm improves the number of processors of the Helmbold and Mayr algorithm for the problem. However, the complexity bounds are far from optimal: recall that the sequential algorithm given in [5] uses time $O(e + n\alpha(n))$, where e is the number of edges in the precedence graph and $\alpha(n)$ is an inverse Ackermann's function. We suspect that such an optimal algorithm must have quite a different approach, in which the levelling algorithm is not used.

Interestingly enough we have shown that computing the lexicographically first matching for full convex bipartite graphs is in NC; in contraposition with the results given in [1] which show that many problems defined through a lexicographically first procedure in the plane are P-complete. We conjecture that all these problems fall in NC when they are convex.

References

[1] M. Attallah, P. Callahan, M. Goodrich, P-complete geometric problems, Internat. J. Comput. Geom. Appl. 3 (4) (1993) 443–462.
[2] E.G. Coffman, R.L. Graham, Optimal scheduling for two processors systems, Acta Inform. 1 (1972) 200–213.
[3] E. Dekel, D. Nassimi, S. Sahni, Parallel matrix and graph algorithms, SIAM J. Comput. 10 (1981) 657–675.
[4] M. Fujii, T. Kasami, K. Ninomiya, Optimal sequencing of two equivalent processors, SIAM J. Comput. 17 (1969) 784–789.
[5] H.N. Gabow, An almost linear time algorithm for two processors scheduling, J. ACM 29 (3) (1982) 766–780.
[6] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the theory of NP completeness, Freeman, San Francisco, 1979.
[7] D. Helmbold, E. Mayr, Two processor scheduling is in NC, SIAM J. Comput. 16 (4) (1987).
[8] J.D. Ullman, NP-complete scheduling problems, J. Comput. System Sci. 10 (1975) 384–393.
[9] U. Vazirani, V. Vazirani, Two-processor scheduling problem is in random NC, SIAM J. Comput. 18 (4) (1989) 1140–1148.