October 1985

Report No. STAN-CS-85-1079

Two Processor Scheduling is in NC

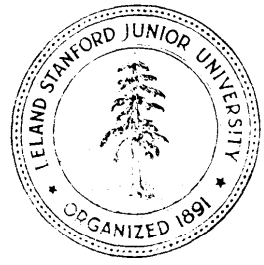
by

David t Ielmbold

Ernst Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305



Stanford University
Department of Computer Science

Two Processor Scheduling is in NC

by

David Helmbold and Ernst Mayr

Abstract: We present a parallel algorithm for the two processor scheduling problem. This algorithm constructs an optimal schedule for unit execution time task systems with arbitrary precedence constraints using a polynomial number of processors and running in time polylog in the size of the input. Whereas previous parallel solutions for the problem made extensive use of randomization, our algorithm is completely deterministic and based on an interesting decomposition technique. And it is of independent relevance for two more reasons. It provides another example for the apparent difference in complexity between decision and search problems in the context of fast parallel computation, and it gives an NC-algorithm for the matching problem in certain restricted cases.

This work was supported by ONR contract N000 1.4-85-C-073 1 and NSF grant DCR-8351757.

1. Introduction

This paper presents results on parallel algorithms for scheduling problems. Our main result is a deterministic $\mathcal{NC}$ algorithm for solving the two processor scheduling problem. This problem falls into the general class of unit time scheduling problems with precedence constraints. The prcccdcncc constraints arc given as a partial order on the tasks; if task t preceeds task t' (written $t \prec$ t') then t must be completed before t' can bc started. A solution to the problem is an optimal (i.c. shortest length) schedule indicating when each task is started.

Several results follow immediately from our main result. Since any optimal schedule corresponds to a maximum matching in the complement of the preccdcncc graph, we have an $\mathcal{NC}$ algorithm which finds maximum matchings in the complements of prcccdcnce graphs (e.g. interval graphs and permutation graphs). For more details, see [HM85] and [KVV85]. In addition, our algorithm solves the "obvious" open problem stated in [VV85].

For research into parallel algorithms, the two processor case is the most interesting unit time sched uling problem. When only a single processor is available (tasks must bc scheduled one at a time) finding an optimal schedule is trivial. If the number of processors is an input to the problem, then the unit time scheduling problem becomes $\mathcal{NP}$-complete [Ul75]. It is unknown whether or not there is a tractable sequential solution for a fixed number of processors greater than two.

The two processor scheduling problem has a long history and rich literature. The first polynomial time solution, $O(n^4)$, was published by Fujii, Kasami and Ninamiya in 1969 [FKN69]. Three years later, Coffman and Graham published an $O(n^2)$ algorithm [CG72]. Gabow found an algorithm that, when combined with Tarjan's union-find result [GT83], runs iii $O(n + e)$ time [Ga82] and hence i s asymptotically optimal. The only pu blished parallel algorithm for the problem is Vazirani and Vazirani's randomized parallel solution [VV85]. The expected running time of their algorithm is a polynomial iii the logarithm of the number of tasks.

Sevcrnl researchers have consitlcred restricting the prcccdcncc constraints and allowing the number of processors to vary. If the precedence constraints arc rcstrictcd to forests [Hu61, HM84, DUW84] then optimal schedules can be found either sequentially or in parallel. If the precedence constraints arc restricted to interval orders then there is a sequential polynomial time algori thm for the problem [PY 79].

With the rising use of highly parallel computers, it is important to identify those problems which can bc cfliciently solved in parallel. It is generally accepted that those problems in the class $\mathcal{NC}$ (solvable in poly-log time using a polynomial number of processors) are amenable to paralellization while those that are $\mathcal{P}$-complete (polynomial time complete under log-space reduction) arc not. The class $\mathcal{RNC}$ consists Of those problems solvable in poly-log time using a polynomial number of processors *with high probability* when random coin flips arc available as a basic computation step. Algorithms in this class only need to

quickly obtain the right answers most of the time.

One fundamental problem which has an $\mathcal{RNC}$ (but no known $\mathcal{UC}$) algorithm is the matching problem [KUW85a]. Our results on two processor scheduling provide evidence that the matching problem might be in UC, since the two problems are closely related. An optimal two processor schedule is a maximum matching in $\bar{G}$, the complement of the precedence graph. Conversely, there is a scqucutial algorithm for convcrtirlg any maximum matching in $\bar{G}$ into an optimal schedule for G [FKN69].

Our $\mathcal{NC}$ algorithm for the two processor scheduling problem is an improvement over the aforementioned $\mathcal{RNC}$ result for two reasons. First, the algorithm in [VV85] is a randomized algorithm; even though the expected running time is poly-log, it may take an arbitrarily long time to halt. Secondly, their algorithm heavily relies on powerful $\mathcal{RNC}$ subroutines for computing maximum matchings and node sets of maximum matchings. In contrast, our algorithm is deterministic, does not require a matching subroutine, and contains interesting parallel techniques such as recursive decomposition.

Because our two processor scheduling algorithm is complex, WC have divided its presentation into several stages. Our first algorithm simply computes the length of au optimal schedule. Using this algorithm WC can locate the "empty slots': or holes in *lexicographically maximum jump* (LMJ) sd1edul es. This in turn enables us to find the lexicographically maximum jump sequence. The most complicated part of our presentation tlcscribes how tasks are assigned to jumps in the jump sequence. Once we have assigned tasks to each jump, it is easy to compu tc an optimal schedule.

2. Basic Definitions

We use the partial order on the tasks to define other useful quantities. A pair of tasks is *independen t* if neither prcccclcs the other. The *precedence graph*, G, is the transitively closed directed acyclic graph with nodes representing the tasks and an edge going from t to t' iff $t \prec t'$. We say task t belongs to level l if the longest path in G from t to a sink contains l nodes (counting both t and the sink). We use "lcvcl l" to denote the set of tasks on the l^{th} level and L to denote the number of levels in G. The length of an optimal schedule for G is denoted by "OPT(G)". WC say a schedule S has an *empty slot* at some timestep if S maps only one task to that timestep.

A *level schedule* schedules the tasks giving preference to tasks on higher lcvcls. More prcciscly, suppose levels I,, . . . , $l+1$ have already been scheduled and there arc k unscheduled tasks remaining on level 1. If k is even we pair the tasks with each other. If k is odd wc pair $k-1$ *of* the tasks with each other and the remaining task t may (but not necessarily) be paired with a task from a lower lcvcl $1' < 1$.

Definition **Jump:**

Given a level schedule, we say level l **jumps** to level $l' \leq l$ if the last **timestep** containing a task **from** level l also contains a task **from** level I'. If the last task from level l is schcdulcd with an **empty** slot, we say that l jumps to **level** 0. The *actual jump* from lcvcl l is an ordered pair of tasks, (t, t') where t is the last task schcdulcd from lcvcl l and t' is the task scheduled with t.

The *jump sequence* of a **level** schedule is the list of levels jumped to. The *actual jump sequence* is the list of actual **jumps.**

Definition LMJ:

The *Lexicographically Maximum Jump (LMJ)* sequence is the jump sequence (resulting from some lcvcl schcdulc) that is lexicographically greater than any other **jump** sequence resulting **from** a **level** schedule. An *LMJ schedule* is a level schedule whose **jump** sequence is the LMJ sequence.

Theorem **[Ga82]:**

Every LMJ schedule is **optimal.**

A trivial consequence of the definition is that every LMJ schedule for G has the same number of tasks remaining on each level after each timestep. Note that our definition of LMJ schedule is equivalent to the definition of highest level first schedule in [VV85] and [Ga82]. Throughout the remainder of this paper we restrict our attention to LMJ schedules.

3. Computing the Length of an Optimal Schedule

Our algorithm for computing the length of an optimal schedule works by computing the number of timesteps that must intcrvcnc between any two tasks. To get the length of a schedule for some prcccdcncc graph G wc add two new tasks, t_{top} and t_{bot}, such that t_{top} is a predecessor and t_{bot} a successor of all tasks in G. Using the new graph, the number of timesteps that must intervene between t_{top} and t_{bot} is precisely OPT(G).

Definition $D(t, t')$**:**

The *schedule distance* between tasks t and t', $D(t, t')$, is the number of timesteps required to schedule all tasks that are both successors of t and predecessors of t'. If $t \not< t'$ then $D(t, t')$ is -00.

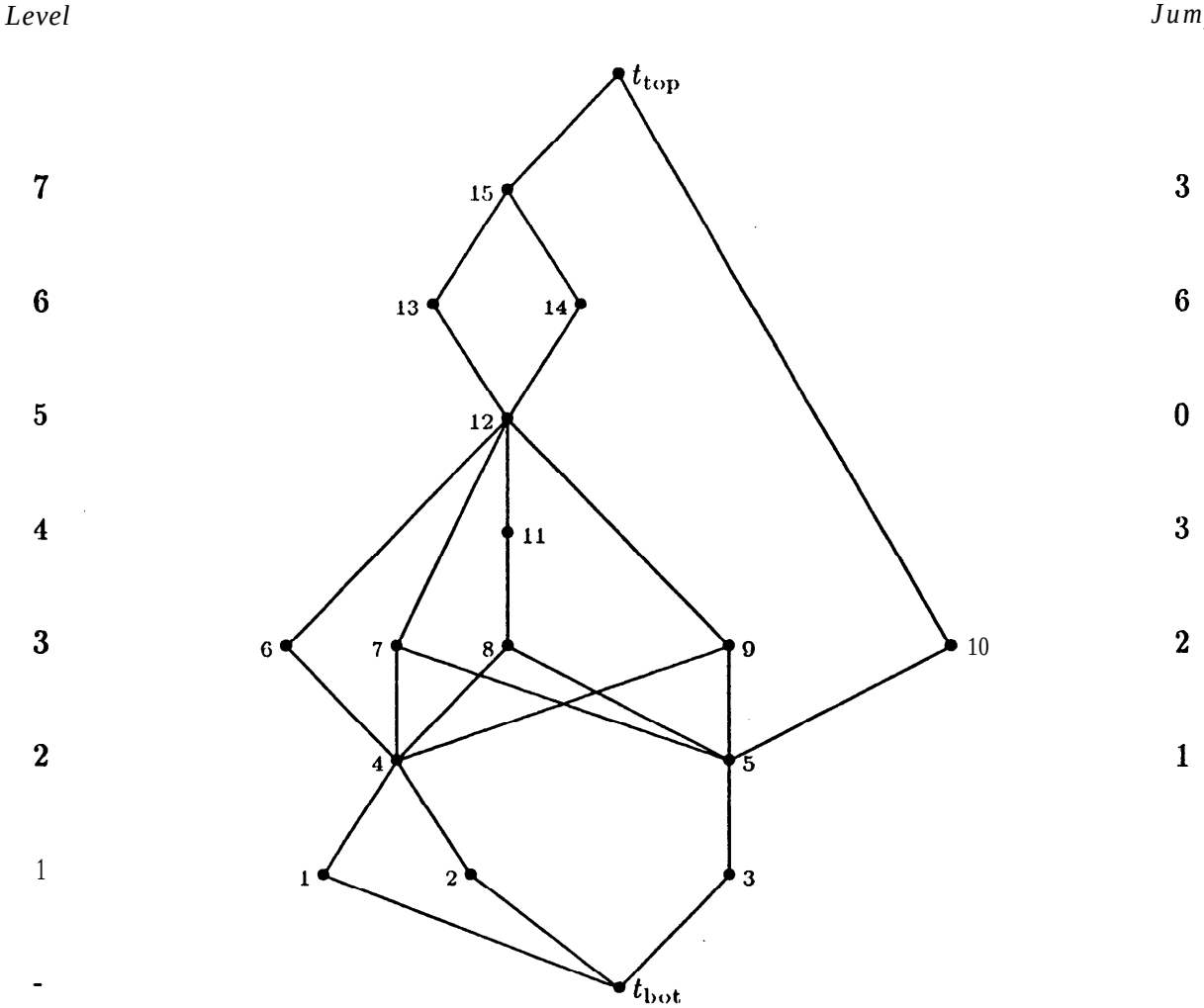


Figure 1

Here is a precedence graph G containing tasks one through fifteen. All precedence constraints arc directed downward. The spccinl tasks t_{top} and t_{bot} are added when computing the length of G's optimal schedules. The levels of the original graph arc on the left and the jump sequence is on the right. Note that the transitive edges have been omitted from the figure.

t_{top}	15	14	12	11	8	6	4	2	t_{bot}
	10	13		9	7	5	3	1	
	χ_5	χ_4		χ_3			χ_2	χ_1	

Figure 2

This is a lexicographically maximal jump schedule for the graph in figure 1. Each of the sets χ_i is boxed. Note that some tasks belong to no χ_i, and that all tasks in χ_i must be completed before those in χ_{i-1} can be stnrtcd.

Lemma 1:

Let t and t' bc any two tasks. If **there** is are **integers** i, k, **and** a non-empty **set** of tasks S such that for all s $\in$ S:

D(t, s) $\geq i$ and

$D(s, t') \geq k$,

then: $D(t, t') \geq i + k + \left\lceil |S|/2 \right\rceil$ (see Figure 3).

Proof: When scheduling the tasks **between** t and t', $t \prec t'$, **there** must bc at least i timestcps **before** the first task in S is **scheduled, and** at lcast k **timesteps after** the **last** one. The set S cannot be **scheduled** in fewer than $\lceil |S|/2 \rceil$ **timesteps,** so it takes at **least** $i + k + \lceil |S|/2 \rceil$ **timesteps** to schedule all the tasks **between** t **and** t'. □

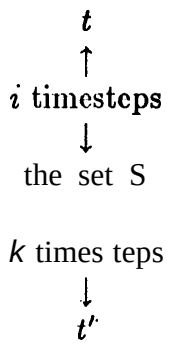


Figure 3

The following algorithm takes a **precedence graph** G and **computes** the **length** of an **optimal** schedule for G.

Algorithm 1:

$d_0(\star, *) := 0$;
for $i := 1$ **to** $\lceil \log n \rceil$ **do**
  for all t, t' with $t \prec t'$ **do** *in parallel*
    for all $0 \leq k$, $l < n - 1$ **do** *in parallel*
      $S_{t,t',k,l} := \{s : t \prec s \prec t', d_{i-1}(t, s) \geq k, d_{i-1}(s, t) \geq l\}$;
      $d_i(t, t') := \max_{k,l,S_{t,t',k,l} \neq \emptyset} \{k + l + \left\lceil |S_{t,t',k,l}|/2 \right\rceil, d_{i-1}(t, t')\}$
$d(\star, \star) := d_{\lceil \log n \rceil}(\star, *)$
$\mathrm{OPT}(G) := d(t_{\mathrm{top}}, t_{\mathrm{bot}})$

Algorithm 1 has a straightforward **implementation** on au n^5 **processor** P-RAM taking $O(\log^2 n)$ **time.**

Lemma 2:

Algorithm **1** never computes a distance between two nodes larger than the schedule distance.

Proof: The algorithm computes each distance by the method in Lemma **1.** Since this method always gives a lower bound on the schedule distance, the distances computed by the algorithm will never exceed the schedule distance. □

In our proof that algorithm **1** computes the proper distance, we borrow a result from [CG72]. There it is shown how to construct sets of tasks $\chi_0, \chi_1, \ldots, \chi_k$ for any precedence graph such that:

1. those tasks in any χ_i are predecessors of all tasks in χ_{i-1}, and
2. OPT(C) = $\sum_i \lceil |\chi_i|/2 \rceil$.

Although our algorithm does not explicitly compute these sets, we use them in the correctness proof.

Lemma 3:

After the main loop has been executed $\lceil \log n \rceil$ times, $d_{\lceil \log n \rceil}(t_{top}, t_{bot})$ is at least $D(t_{top}, t_{bot})$ = OPT(G).

Proof: By induction. Let $d(\chi_i, \chi_k)$ denote the least $d(t, t')$ where $t \in \chi_i$ and $t' \in \chi_k$. After the first iteration of the algorithm,

$$d_1(\chi_l, \chi_{l-2}) \geq \lceil |\chi_{l-1}|/2 \rceil$$

since all members of χ_{l-1} are between any $t \in \chi_l$ and any $t' \in \chi_{l-2}$. Assume, by induction, that after r iterations of the main loop:

$$d_r(\chi_l, \chi_{l-2^r}) \geq \sum_{l-2^r<j<l} \lceil |\chi_j|/2 \rceil \ .$$

We must show, that after the $r + 1^{st}$ iteration of the main loop,

$$d_{r+1}(\chi_l, \chi_{l-2^{r+1}}) \geq \sum_{l-2^{r+1}<j<l} \lceil |\chi_j|/2 \rceil \ .$$

If we let $i = \sum_{l-2^r<j<l} \lceil |\chi_j|/2 \rceil$ and $k = \sum_{l-2^{r+1}<j<l-2^r} \lceil |\chi_j|/2 \rceil$, then after the r^{th} iteration $d_r(\chi_l, \chi_{l-2^r}) \geq i$ and $d_r(\chi_{l-2^r}, \chi_{l-2^{r+1}}) \geq k$. Therefore, after the $r+1^{st}$ iteration:

$$d_{r+1}(\chi_l, \chi_{l-2^{r+1}}) \geq i + k + \lceil |\chi_{l-2^r}|/2 \rceil = \sum_{l-2^{r+1}<j<l} \lceil |\chi_j|/2 \rceil \ . \ \square$$

Theorem 1:

Algorithm **1** correctly computes the length of the shortest schedule for the precedence graph.

Proof: This Theorem follows from Lemmas 2 and 3. []

It is easy to convert a polynomial-time algorithm that computes the length of an optimal schedule to one computing a particular optimal schedule. This is because we can try each possibility for the first timestep, and check if the remainder of the graph can bc scheduled in one less timestep. Unfortunately this process is inherently sequential. We must wait until the first k timesteps are fixed before we attempt to fix the $k + 1^{st}$.

The corresponding method for parallel algorithms is to compute the tasks scheduled at the next timestep for each possible initial schedule. Unfortunately, there are exponentially many possible initial schedules. Therefore we must use more powerful methods for determining the actual schedule.

4. Computing the Jump Sequence

The first step in computing the LMJ sequence for G is to obtain an algorithm for determining which levels jump to level 0. When this algorithm is run on appropriate subgraphs of G, we can find the level jumped to by each level in G.

It is easy to determine if the bottom level jumps to level 0 in an LMJ schedule. WC modify G by adding a new task to the bottom level which depends on all of the tasks on levels 2 through $L(G)$. If the length of an optimal schedule for the modified graph is the same as $OPT(G)$, then every LMJ schedule for G ends with an empty slot (jump to level 0). II' the length of an optimal schedule for the modified graph is greater than the length of an optimal schedule for the original graph then wc know that the bottom level ends with a full timestep (a jump to itself) in LMJ schedules for G. Therefore by either adding or not adding the new task we can ensure that LMJ schedules end with full timcsteps. Furthermore, ensuring that LMJ schedules end with a full timestep in this way does not affect the jumps of other levels, since no task on another Jcvcl can be paired with the added task.

Lemma 4:

Given G, there is an $\mathcal{NC}$ algorithm for finding which levels jump to level 0 in the LMJ sequence.

Proof: WC first determine if the bottom level jumps to level 0, and if so modify G so that LMJ schedules end in full timesteps. For each pair (I, e), $0 \leq l$, e $\leq$ T,(G), WC create $G_{l,e}$. Each $G_{l,e}$ consists of the (modified) G plus e additional tasks. Every task above level l in $G_{l,e}$ has a precedence arc to each of the e additional tasks.

The length of an optimal schedule for $G_{l,e}$ will be the same as the length of the optimal schedule for G only when all of the e added tasks are jumped to by levels that used to jump to level 0. Therefore by determining the largest e such that $\mathrm{OPT}(G_{l,e})$ = OPT(G) we can find the number of levels at or below level l that jump to level 0 in an LMJ schedule for G. Those values of l where this changes are precisely those levels that jump to 0 in an LMJ schedule for G. □

Now WC must convert our algorithm for finding jumps to level 0 into one that finds all of the jumps. Say level l jumps to I', then if level I' and all lower levels were deleted, level l would jump to level 0. This observation leads us to an algorithm for determining the LMJ sequence for a precedence graph.

Lemma 5:

Given a precedence graph G, there is an $\mathcal{NC}$ algorithm for finding the LMJ sequence.

Proof: Find the jumps to level 0 in LMJ schedules for the graphs $G_L, G_{L-1}, \ldots . G_1$ where G_l contains only the nodes on levels L down to l (and the precedence constraints between them). If level l does not jump to level 0 in G_l then (I, 1) is in the LMJ sequence for G. If level l jumps to level 0 in G_1 then $(l, 0)$ is in the LMJ sequence for G. Otherwise, for some $0 < i < 1$, the jump $(l, 0)$ occurs in G_{i+1} but does not show up in G_i, so there is a jump from level l to level i in the LMJ sequence for G. □

5. Candidates for Actual Jumps

After obtaining the jump sequence we must *assign* a pair of tasks to each jump. This gives us the actual jumps. In this section WC present an algorithm for determining whether or not a pair of tasks can be used as an actual jump.

Definition Candidate Pair:

A pair of tasks (t, t') is a *candidate pair* for the jump (l, l') if:

1. t is from l and t' is from l' and
2. (t, t') is an actual jump in some LMJ schedule.

Lemma 6:

Any set of disjoint candidate pairs (one for each jump) can be used as the actual jumps in an LMJ schedule.

Proof: Assume we have a set of disjoint candidate pairs which can not be used as the actual jumps in any LMJ schedule. Then there is a first task, t', which is scheduled before all of its predecessors have been completed. All tasks, except the second tasks in the candidate pairs, are scheduled only after all higher levels have been completed. Therefore, t' must be the second task in some candidate pair, say (t, t') for the jump (l, l'). WC know

that t and t' arc independent (since they are scheduled together in some LMJ schedule) and t' has an unscheduled predecessor on a level between l and *1'*. That predecessor of t' is independent with t, so a better jump than (l, l') is possible. Therefore we did not start with an LMJ sequence -- contradiction. []

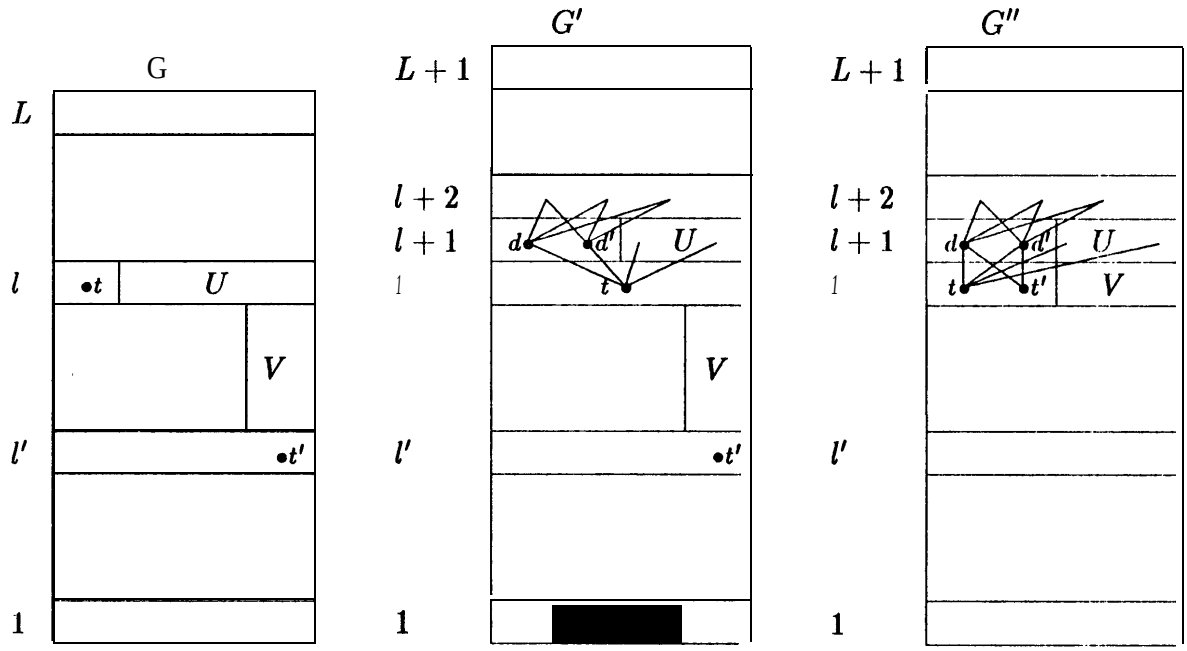


Figure 4

WC USC this construction to check if the jump ($t \in$ I, $t' \in$ I') is part of any LM J schedule. The set U contains the tasks on level l, except for t. The set V contains those tasks on levels between 1 and 1' which are *not* successors of *1*.

Assume that level *1* jumps to *1'* in LMJ sequences for G and that $t \in 1$ and $t' \in 1'$ are independent. Tt is important to know whether or not any LMJ schedule for G maps t and t' to the same timestep. By modifying the prcccdcncc constraints of G we can answer this question.

Figure 4 shows the modifications we use. The graph G is the original prcccdeucc graph. The set U consists of all tasks on level l, except for task t. The sot V consists of those tasks on levels between l and l ' which are *not* successors of t. I f t and t' are mapped to t h e same timestep in some LMJ schedule, then t is the last task schedu led from level *1*. The first part of the construction checks that t is not needed for a jump from a higher lcvcl and that t can bc paired with some task from level *1'*:

The graph G' is created from G by the following procedure:

1. Create two dummy tasks, d and d'.
2. Make t a successor of d, d', and the tasks in U.
3. Make d and d' successors of all tasks on level *1* + 1 of G.

The effect of this procedure is to "move up" level 1 and every higher level. The new level l contains only task t. The level $l + 1$ in G' contains those tasks that were in level l (except for t) in G and the two dummy tasks. The level $l + 2$ in G' contains exactly those tasks on level $l + 1$ in G and so on. Note that no task can jump to d, d', or t.

Using the algorithms from the previous section, we can compute the LMJ sequences for G and G'. We say the LMJ sequcnccs for G and G' arc *similar* if: the jump $(l + 1, l + 1)$ is in the sequence for G'; and whenever the jump (j, k) is in the sequence for G, then the jump (j', k') is in the sequence for G'. Here we use j' (k') to be j + 1 $(k + 1)$ if j > l $(k > l)$ and j (k) otherwise.

Lemma 7:

If the LMJ sequences for G and G' are similar, then any schedule for G' completes all of the tasks in V before level l is finished.

Proof: Otherwise task t would have jumped to some level above level l' in the LMJ schedule for G'. []

Lemma 8:

There is an LMJ schedule for G where task t is the last task scheduled from level l if and only if the LMJ sequences for G and G' are similar.

Proof: Assume G has an LMJ schedule where t is the last task on level l. By inserting a timestep for the two dummy tasks we get a similar LMJ schedule for G'. Assume the LMJ sequences for G and G' arc similar. Since level l + 1 jumps to itself, and no task can jump to either of the dummy tasks, there is an LMJ schedule, S, for G' which pairs the two dummy tasks at some timestep T. The schedule obtained by deleting timestep T from S is an LMJ schedule for G. []

Now WC use the following procedure to construct G'' from G':

1. Dcictc all precedence constraints bctwccn tasks in V U {t'}.
2. Let s bc some successor of t on level l − 1; make every task in V U {t'} a predecessor of s.
3. Make t' a successor of both d and d'.

The LMJ sequences for graphs G' and G'' are *similar* if: the jump (l, l') is in the sequence for G'; the jump (I, l) is in the sequence for G"; when the jump (j, k), j > I, $k \in [l'+1, l-1]$, is in the scqucncc for G', the jump (j, l) is in the sequence for G"; and ail other jumps in G' arc also in G".

Lemma 9:

If there is an LMJ schedule for G' which maps tasks t and t' to the same timestep, then the LMJ sequences for G' and G'' are similar.

Proof: Such a schedule for G' is also an LMJ schedule for G", therefore the LMJ sequences for G' and G'' are similar. □

Lemma IO:

If the LMJ sequences for G' and G'' are similar, then there is an LMJ schedule for G' which maps t and t' to the same timestep.

Proof: Assume that the LMJ sequences for G' and G'' arc similar and no LMJ schedule for G' maps t and t' to the same timestep. We know that in some LMJ schedule for G' t is the last task from level 1, t is scheduled with a task from level l', and that t and t' are independent. Therefore, t' must be needed for some other jump. Since all of the other jumps to/from level l' in G' are also present in G", t' is also needed for one of them in G". However, t' is paired with t in G'' – contradiction. □

Theorem 2:

There is an $\mathcal{NC}$ algorithm to determine whether the pair (t, t') is a candidate for the jump from level l to I'.

Proof: Create the graphs G' and G'' and check that the LMJ sequences of G and G', and of G' and G'' arc similar. □

The above Theorem leads us to the obvious algorithm for assigning tasks to a particular jump in the LMJ sequence. We simply try all independent pairs of tasks from the appropriate levels and pick any pair that satisfies Theorem 2. This method assures us that there is at least one valid way of assigning tasks to the remaining jumps. However, WC still need powerful tools in orticr to pick consistent pairs rapidly in parallel.

6. Parallel Selection of Candidate Pairs

In the previous section WC presented a method for determining if a pair of tasks from different levels is a candidate pair. From Lemma 6 we know that any disjoint set of candidate pairs (one for each jump in the LMJ sequence) forms the actual jumps of an LMJ schedule. Therefore our problem is to pick a candidate pair for each jump in the LMJ sequence while guaranteeing that no task i s in more than one pair.

This problem is similar to the matching problem, so powerful techniques arc needed it solve it in parallel. On the other hand, since each level jumps to at most one other icvel, the jumps form a tree structure. WC exploit this structure using *recursive decomposition.* At each stage in the decomposition we split the current problems into two or more disjoint problems. When all of the problems have been reduced to finding a single candidate pair, they can be solved quickly in parallel.

Definition Jump **Forest:**

The jump *forest* for a graph G contains one node for **each level** in G. An arc goes from the node for **level** l to the **node** for level 1' $\neq l$ if and only if the jump (l, l') is in the LMJ **sequence** for G.

This **structure** is an **inforest** since at most one jump **originates** from each level. Note that jumps to the same level and jumps to level 0 are not **represented** in the jump tree.

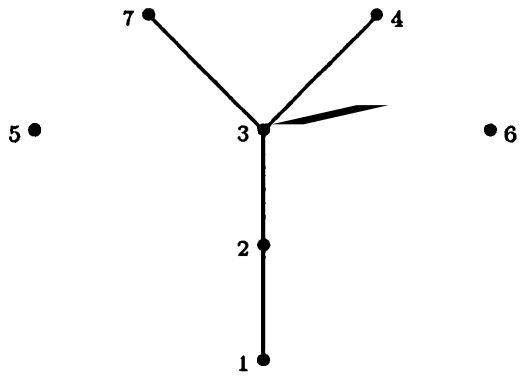


Figure 5

This is the jump forest for **the precedence graph** in Figure **1. Each node** represents **an entire** level in the precedence graph. All edges (jumps) arc directed downwards.

Lemma 11:

If **the** jump **forest** consists of two or **more** disjoint trees, **then candidate** pairs can be **picked** for **each** tree **independently.**

Proof: Disjoint **trees** do not share **levels, hence they** do not sharc tasks. Thcreforc no task in a candidate pair for **one tree** can also bc in a candidate pair for the **other tree.** []

For each jump **tree** J in **the forest, there** is **an induced subgraph** of G. **The induced** subgraph, G_J, **contains those levels of** G which **are represented** in J. **Thus each level** of G is in cxnctly **one induced subgraph. Because the jump trees** arc split at **each** stage of **the algorithm,** it is **convenient** to **denote levels in induced subgraphs** by **their names** in G **rather** than **renumbering** them **every step. Initially each induced subgraph contains** all the jobs from its **constituent levels** (and **the precedence** constraints **between them).**

As a jump **tree** is split, **certain candidate** pairs will be **picked. In** order to **keep the** tasks in **these candidate** pairs **from being reused,** we **delete them from the induced subgraphs.** We **continue** splitting jump **trees (and induced subgraphs) until each jump tree** consists of

a single level. All tasks remaining in the induced subgraph can then bc arbitrarily paired together since there are never precedence constraints between tasks on the same level.

Lemma 12:

Any intree of size n can be split into disjoint subtrees, each of size at most n/2, by removing a single node. Furthermore, an appropriate node can be found quickly in parallel.

Proof: Count the number of ancestors for each node. When the highest node with at least n/2 ancestors is removed, the remaining components will all have size at most $n/2$. []

Since we halve the size of the largest jump tree each iteration, after at most log n iterations all of the jump trees will consist of a single level.

The first step to remove an internal level 1 from a jump tree J is to make 1 the root. Using G_J WC can find a candidate pair (t, t') for the jump (l, l'). If we delete t and t' from G_J, the jump (I, 1) is no longer in G_J's LMJ sequence and J is split. We use J(1) to denote the part of J rooted at 1 and $J(l')$ to denote the part containing $1'$.

Unfortunately, when WC delete t and t' the level structure of our induced subgraphs may be dcstroycd. Some tasks in $G_{J(l)}$ may have preceeded only t from level l. We must adjust the precedence constraints of $G_{J(l)}$ so that these "dangling tasks" preceed at least one task on level l. Similarly, we must adjust the precedence constraints of $G_{J(l')}$ so that tasks which used to preceeded l' still preceed a task on level $1'$.

Level 1 is the root of $J(l)$. Therefore WC can add a dummy task to level 1 which is a successor Of all tasks in 1110 jump Crcc, except those on level l. Now every task in $G_{J(l)}$ is either on level l or preceeds a task from Icvcl l. If l' is the root of' $G_{J(l')}$, *then* we can use the same method.

If l' is not the root of its jump tree, then it jumps to some other level. Let $\hat{t} \in l'$ be any task which is not needed for another level's jump to $1'$. We know there is at least one such task, namely the one used for the jump from lcvcl $1'$. The level property is restored when all of the tasks in $G_{J(l')}$ on levels above l' are made predecessors of $\hat{t}$. The added precedence constraints do not affect the LMJ schedule for $G_{J(l')}$ since all tasks above level l are completed before $\hat{t}$. WC can find a suitable $\hat{t}$ by trying all tasks on level l' in parallel and selecting any one where the added precedence constraints do not change the LMJ sequence of $G_{J(l')}$.

Now level 1, the one to bc removed, is at the root of its jump tree. Let levels $l_1, l_2, l_3, \ldots, l_m$ be the levels in the jump tree which jump directly to l in *ascending order*. Thus l_m is the highest level which jumps to level l. Our algorithm assigns each of the levels l_1 through l_m a particular task to jump to.

Definition Chunk, C_i:

The chunks are sets of tasks from level 1. For $1 \leq i \leq m$, C_i consists of those tasks from level l which are in a candidate pair for a jump (l_j, l) where $j \geq i$.

The set C_i may be larger than the set of tasks from level l which can be used for the jump, (l_i, l). It may include tasks which must be used for other jumps. Since there is a way to assign disjoint candidate pairs to the m jumps, $|C_i| \geq m - i + 1$.

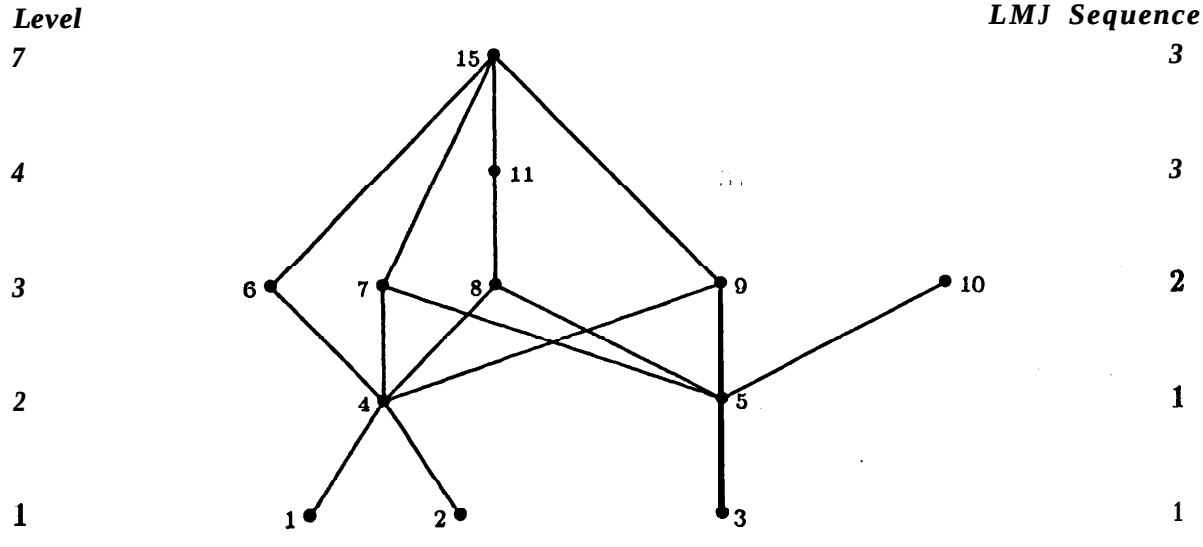


Figure 6a

This is the induced subgraph for the initial jump tree containing level 3. As usual, transitive edges have been omitted from the figure.

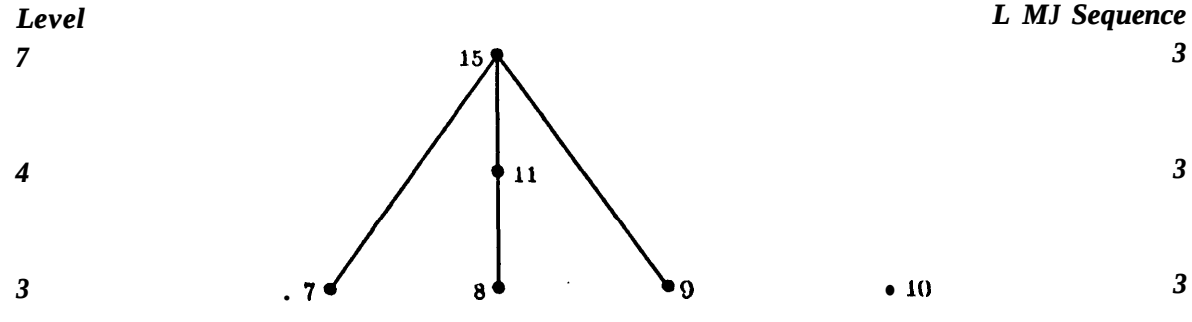


Figure 6b

After the jump $(3, 2)$ has been removed from the tree (using the only candidate pair, tasks 5 and 6), we get a new induced subgraph. Tasks 15 and 11 are the only tasks in candidate pairs on their respective levels, so task 10 is in two chunks and tasks 7 and 9 are in one chunk. Therefore, task 10 will be assigned to level 7 and either task 7 or 9 will be assigned to level 4.

Lemma 13:

If $1 \leq j < i \leq m$ and $t \in C_i$ then all tasks on level l_j are either needed for jumps to level l_j or are independent with t.

Proof: Assume there is some $t_j \in l_j$ for which the Lemma does not hold. Because t is in C_i, there is some task t_i on or above level l_i which forms a candidate pair with t. Since t_j is a predecessor of t, t_i and t_j must be independent. If t_j has not yet been scheduled when t_i's level is completed then tasks t_i and t_j could be paired and t_i's level would jump to level l_j in LMJ schedules for G – contradiction. □

Our algorithm simply sorts the tasks on level l by the number of chunks they are in, breaking ties arbitrarily. The task in the most chunks is assigned to level l_m, the task in the second most chunks is assigned to l_{m-1} and so on.

Lemma 14:

There is always a t_i on level l_i which can be paired with the task assigned to l_i forming a candidate pair.

Proof: All tasks in C_i are in at least i chunks and any task not in C_i can be in at most $i - 1$ chunks. Since there are at least $m - i + 1$ tasks in C_i, the task assigned to level l_i will be in C_i. As a consequence of Lemma 13 and the definition of chunks, there is at least one candidate pair for the jump (l_i, l) containing the task assigned to level l_i. □

Theorem 3:

There is an $\mathcal{NC}$ procedure to remove an arbitrary node from a jump tree, finding disjoint candidate pairs for all of the dclctcd edges.

Proof: Tn order to remove level l from jump tree J we first remove any outgoing jump from l. By Theorem 2, there is an $\mathcal{N}$ C algori thm for doing this. Creating the chunks, C_i can easily bc done in parallel since WC have an $\mathcal{NC}$ algorithm for dcterruining all candidate pairs for a given jump. Sorting is in $\mathcal{NC}$, so we can assign a task to each l_i quickly in parallel. Once task t has been assigned to level l_i we can test all of the pairs $(t_i \in l_i, t)$ in parallel, using any candidate pair for the jump (l_i, l). □

Theorem 4:

There is an NC algorithm which finds a two processor LMJ schedule for any prcccdcncc graph.

Proof: Create the jump forest. Recursivly decompose each jump tree, saving the picked candidate pairs. Arbitrarily pair the remaining tasks on each lcvcl. By sorting these pairs of tasks WC obtain an LMJ schedule. □

Our algorithm is intended to show that two processor scheduling is in $\mathcal{NC}$; it is not intended

to be an efficient solution. Computing the LMJ sequence for a graph G uses $L(G)^2$ calls to the schedule length algorithm (in parallel). Determining if a pair of tasks is a candidate pair basically involves computing two LMJ sequences. Thus the total requirements for computing all of the candidate pairs is $n^2L(G)^2n^5$ processors and $O(\log$ n log L(G)) time. If the candidate pairs have been pre-computed, the cost of decomposing the jump trees becomes insignificant. Therefore, the requirements for the entire algorithm are $n^7L(G)^2$ processors and O (log n log L(G)) .

7. Conclusions

Our two processor scheduling result, coupled with a transitive orientation algorithm, allows us to solve several restrictions of the maximum matching problem. By exploiting the special relationship between two processor scheduling and matching, we can find maximum matchings on undirected graphs whose complements have a transitive orientation. Specifically, if G is an interval graph, then $\bar{G}$ has a transitive orientation [Gh62]. Similarly, G is a permutation graph if and only if both G and $\bar{G}$ are transitively orientable [PLE71]. Thus we have a deterministic $\mathcal{NC}$ maximum matching algorithm for both interval graphs and permutation graphs [HM85, KVV85].

There are many variations of the fundamental two processor scheduling problem. The tasks can have varying execution times, release times, or deadlines. If tasks have small integer execution times and are preemptable, then the problem reduces to the unit time scheduling problem. So far, our attempts at extending the algorithm to handle deadlines and/or release times have been unfruitful.

One variant of the two processor problem that WC know to be N $\mathcal{P}$-complete (under log-space reduction) allows incompatability edges as well well as precedence constraints. When there is an incompatability constraint between two tasks they can be executed in either order, but not concurrently. Incompatability constraints arise naturally when two or more tasks need the same resource, such as special purpose hardware or a database file.

It was surprising how much more difficult computing the actual schedule was than simply computing its length. In higher complexity classes such as $\mathcal{P}$ and $\mathcal{NP}$ it is often easy to go from the decision problem to computing an actual solution, because of self reducibility. Ilowcvcr this does not seem to necessarily bc the case for parallel complexity classes. To support this observation we note that the random NC algorithm for finding the cardinality of a maximum matching is much simpler than the random NC algorithm for determining an actual maximum matching [KUW85b].

References:

[CG72] Coffman, E.G., Jr., and R.L. Graham "Optimal Scheduling for Two Processor Systems," Acta Informatica 1, (1972) 200-213.

[DUW84] Dolev, D., E. Upfal, and M. Warmuth, "Scheduling Trees in Parallel," Proc. International Workshop on Parallel Computing and VLSI, Amalfi, (1984) 1-30.

[FKN69] Fujii, M., T. Kasami, and K. Ninamiya, "Optimal Sequencing of Two Equivalent Processors," SIAM J. of Computing 17 (1969) 784-789.

[Ga82] Gabow, H.N., "An Almost-linear Algorithm for Two-processor Scheduling," JACM, 29, 3 (1982) 766-780.

[GT83] Gabow, H.N. and Tarjan, R.E., "A Linear Time Algorithm for Special Case of Disjoint Set Union", proc. 15th STOC (1983).

[Gh62] Ghouilà-Houri, A., "Caractérisation des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre," C.R. Acad. Sci. Paris 254 (1962).

[HM84] Helmbold, D. and E. Mayr, "Fast Scheduling Algorithms on Parallel Computers," Stanford tech. report STAN-CS-84-1025, (1984).

[HM85] Helmbold, D. and E. Mayr, "Transitive Orientation and NC Algorithms," in preparation.

[Hu61] Hu, T.C., "Parallel Sequencing and Assembly Line Problems," Operations Research 9 (1961) 841-848.

[KUW85a] Karp, R.M., E. Upfal, and A. Wigderson, "Constructing a Perfect Matching is in Random NC," proc. 17th STOC (1985).

[KUW85b] Karp, R.M., E. Upfal, and A. Wigclcrson, "Arc Search and Decision Problems Computationally Equivalent?," proc. 17th STOC (1985).

[KVV85] Kozen, D., U.V. Vazirani, and V.V. Vazirani, "NC Algorithms for Comparability Graphs, Interval Graphs, and Testing for Unique Perfect Matching," to appear.

[VV85] Vazirani, U.V. and V.V. Vazirani, "The Two-Processor Scheduling Problem is in RNC," proc. 17th STOC (1985).

[PY79] Papadimitriou, C.H. and Yannakakis, M., "Scheduling Interval-Ordered Tasks," SIAM J. Computing vol 8 #3 (1979).

[PLE71] Pnueli, A., A. Lempel, and S. Even, "Transitive orientation of Graphs and Identification of Permutation Graphs," Can. J. Math., vol 23 #1 (1971) 160-175.

[Ul75] Ullman, J.D., "NP-complete Scheduling Problems," J. Comput. System Sci. 10 (1975), 384-393.