T U M

I N S T I T U T F Ü R I N F O R M A T I K

Optimal Parallel Algorithms for Two Processor Scheduling with Tree Precedence Constraints

Ernst W. Mayr Hans Stadtherr

TUM-I9531
November 1995

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-1995-I9531-300/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1995 MATHEMATISCHES INSTITUT UND
INSTITUT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Optimal Parallel Algorithms for Two Processor Scheduling with Tree Precedence Constraints

Ernst W. Mayr   Hans Stadtherr

Institut für Informatik
Technische Universität München
80290 München, Germany
{mayr|stadther}@informatik.tu-muenchen.de

November 8, 1995

Abstract

Consider the problem of finding a minimum length schedule for n unit execution time tasks on m processors with tree-like precedence constraints. A sequential algorithm can solve this problem in linear time. The fastest known parallel algorithm needs $O(\log n)$ time using n^2 processors. For the case $m = 2$ we present two work optimal parallel algorithms that produce greedy optimal schedules for intrees and outtrees. Both run in $O(\log n)$ time using $n/\log n$ processors of an EREW PRAM.

Keywords: parallel algorithms, scheduling, tree precedence constraints, optimal work

1 Introduction

The need to utilize resources economically gives scheduling theory an important role in computer science. Another reason for the popularity of scheduling is its relevance in complexity theory. Many if not most scheduling problems are intractable, i.e. they belong to the class of $\mathcal{NP}$-hard problems. The border between intractable scheduling problems and those for which efficient algorithms are known is a challenging area for investigations. Consider the problem of finding a schedule for n equal length tasks constrained by an arbitrary precedence relation such that the total time needed to execute all tasks on m identical processors is minimized. Ullman has shown that this problem is $\mathcal{NP}$-complete in general [Ull75]. If we restrict the problem to $m = 2$ it becomes polynomially solvable [FKN69, CG72, Gab82] and even has an efficient parallel (i.e., $\mathcal{NC}$) algorithm [HM87b].

In this paper we restrict our attention to tree-like precedence constraints. This class of scheduling problems has attracted special interest since the 1960s originating in expression evaluation and assembly line production problems. An early result by Hu [Hu61] shows that unit execution time tasks constrained by a tree precedence relation can be optimally scheduled for an arbitrary number of processors in polynomial time. Brucker, Garey, and Johnson later showed that the problem can even be solved by a linear time algorithm [BGJ77]. It is interesting to note that slight generalizations of the tree precedence structure, e.g. one outtree combined with one intree, result in intractable problems [May81, DUW86].

We are interested in the parallel complexity of scheduling equal length tasks with tree precedence constraints. Helmbold and Mayr [HM87a] developed two EREW PRAM algorithms that compute greedy schedules for intrees and outtrees. Both run in $O(\log n)$ time using n^3 processors. Dolev, Upfal and Warmuth [DUW86] reduced the problem of scheduling outtree precedence constraints to finding a perfect matching in a convex bipartite graph which can be solved on an EREW PRAM in $O(\log^2 n)$ time using n processors [DS84]. In addition they developed an algorithm running in $O(\log n)$ time using n^2 processors. An open problem is whether the number of required operations can be lowered. In this paper we show that for the case $m = 2$ work optimal PRAM algorithms exist that compute greedy optimal schedules for intrees and outtrees.

2 Preliminaries

A unit execution time (UET) 2-processor scheduling problem $(T, \prec)$ consists of a set of n tasks T and a partial order $\prec$ on the set of tasks. A solution to the problem is a *schedule* $S : T \to \mathbb{N}^*$ mapping tasks to integer timesteps with the following properties:

- the precedence constraints are satisfied, i.e. if $t \prec t'$ then $S(t) < S(t')$ and
- no more than 2 tasks are mapped to the same timestep.

We assume that schedules always start at timestep 1 (which is the interval $[0, 1)$). The *length* of a schedule S is then $\max_{t \in T}\{S(t)\}$. We are interested in schedules that have minimum length under all possible solutions to a given scheduling problem. Such schedules are called *optimal*.

The *precedence graph* for $(T, \prec)$ is a directed acyclic graph $G = (T, E)$ with vertex set T and edge set $E = \{(t, t')|t \prec t'\}$. We call G a *reduced precedence graph* if no transitive edges are present, i.e. if $t \prec t' \prec t''$ implies $(t, t'') \notin E$. Let depth(G) be the number of vertices on a longest path in G, let depth(t), $t \in T$, be the number of vertices on a longest path ending at t and let height(t) be the number of vertices on a longest path starting at t. Define the *LPT value* of a task t as $\mathrm{LPT}(t) := \mathrm{depth}(G) - \mathrm{height}(t) + 1$. Tasks having the same LPT value are said to be on the same *LPT level*. Intuitively, LPT(t) is the latest possible timestep a task t can be scheduled in order to yield an optimal schedule for an unlimited number of processors. The earliest possible timestep a task t can be scheduled, called its *EPT value*, is obviously depth(t). For a given schedule a task t is said to be *available* at timestep s if all tasks t' with $t' \prec t$ are mapped to timesteps earlier than s. A timestep s is *full* if two tasks are mapped to s, otherwise s is called *partial*. A schedule is *greedy* if there is only one task available at any partial timestep.

In the sequel we will focus our attention on UET 2-processor scheduling problems, where the reduced precedence graph (T, E) is either an *intree* or an *outtree*. In the former case every vertex except one has outdegree 1, and the remaining vertex, called the root, has outdegree 0. In the latter case, every vertex other than the root has indegree 1, and the root has indegree 0. It is well known that a successful strategy for precedence graphs having tree structure is to schedule tasks with lower LPT values earlier than others whenever possible. This strategy is known as the *level strategy*:

Definition 1 *A schedule $S : T \to \mathbb{N}$ is a* level schedule *if it is greedy and there do not exist tasks t and t' such that $LPT(t) > LPT(t')$, $S(t) < S(t')$ and t' is available at timestep $S(t)$.*

*$\mathbb{N}$ denotes the set $\{1, ...\}$ of positive natural numbers.

Theorem 1 *Level schedules are optimal for tree precedence constraints.*

Proof: See [Hu61] for intrees and [Bru82] for outtrees. □

Note that all results apply to forests as well. If the reduced precedence graph is a forest of intrees, we add a new task that succeeds all roots. After finding a greedy optimal schedule for the resulting intree we remove the last task and thereby obtain a greedy optimal schedule for the forest. Forests of outtrees are handled accordingly.

3 Two Processor Level Schedules for Intrees

Consider a UET 2-processor scheduling problem $(T, \prec)$ where the reduced precedence graph $G = (T, E)$ is an intree. Select a longest path in G and call tasks on this path *backbone* tasks. All other tasks in T are said to be *free*. We start with two observations on tasks with equal LPT value and the relationship between free and backbone tasks:

Lemma 1 *Let $t, t' \in T$ be on the same LPT level. Then neither $t \prec t'$ nor $t' \prec t$.*

Lemma 2 *For any two tasks $t, t' \in T$ the following holds: if t is a backbone task and t' a free task with LPT(t) $\leq$ LPT(t') then neither $t \prec t'$ nor $t' \prec t$.*

Thus we have reason to call non-backbone tasks free. We can pair a free task with a backbone task on a lower LPT level without having to fear that precedence constraints are violated. Thus we implement the level strategy by pairing each backbone task that has no free task on its own level with a free task from a higher LPT level. In order to yield a level schedule we have to maintain the LPT order of free tasks that are paired with backbone tasks on lower LPT levels.

3.1 A Sequential Algorithm

Assume we have determined the LPT value of each task. Initially we arrange tasks from left to right sorted by their LPT value. In Figure 1 tasks are depicted as boxes and tasks with equal LPT value are grouped into a single column. We order tasks from left to right and within a single column from top to bottom. We mainly operate on free tasks so we separate them from backbone tasks which we keep in the bottom row. Contiguous columns with more than one task in each column are called a *block*. Imagine all free tasks of a block being concatenated and threaded on a string like pearls as in Figure 1. Pull the string of each block to the left – starting with the leftmost block – until the first task on each string reaches either the left end or the last free task of the next block to the left. In doing so the number of free tasks in each column is not increased beyond 1. The result is shown in Figure 2 and again we assume the order of tasks from left to right and within a column from top to bottom including backbone tasks. Note that each backbone task is still the last on its level. Finally, we assign pairs of tasks from left to right in the resulting order to consecutive timesteps as long as possible. In Figure 3 the mapping of tasks is indicated by greyscale levels, where two consecutive tasks having the same greyscale are supposed to be mapped to the same timestep. The main steps of the algorithm are captured in the following outline. Let $|P|$ denote the length of a sequence P, $P[i]$ the i-th element, and $P[i, ..., j]$ the subsequence starting at $P[i]$ up to and including $P[j]$.

1. Determine the LPT value of each task.

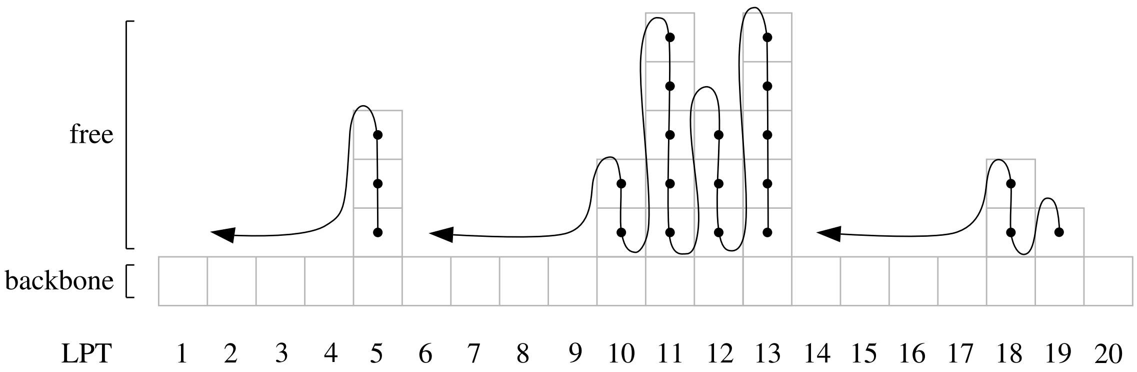

Figure 1: *Tasks ordered by their LPT value. Free tasks in each block are concatenated and threaded on a string.*

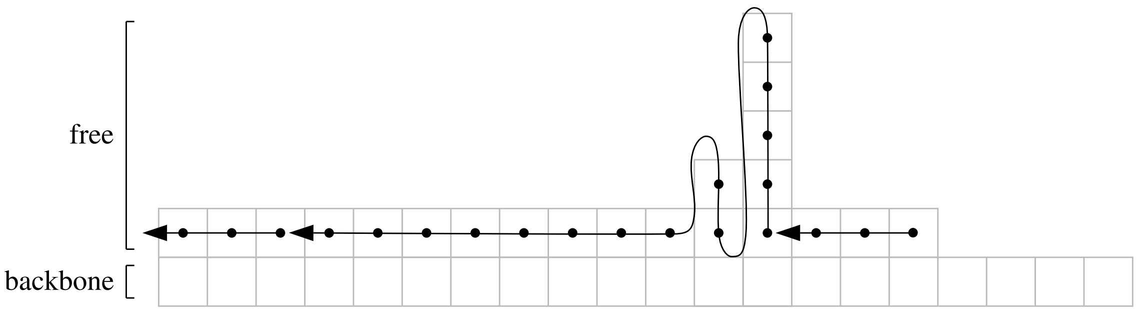

Figure 2: *After pulling the strings to the left.*

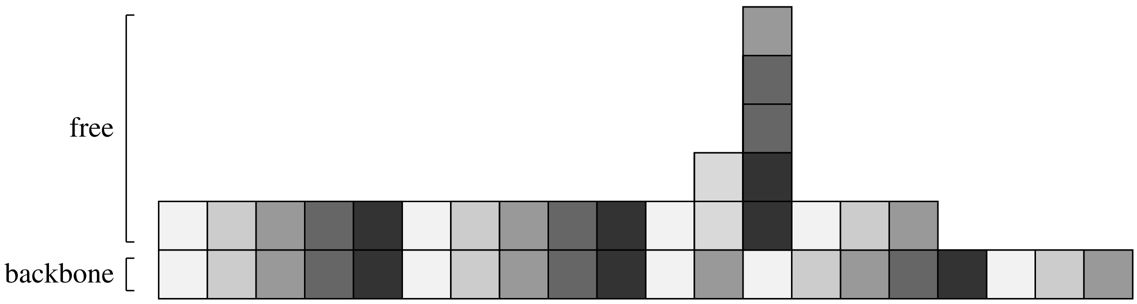

Figure 3: *Pairs of tasks from left to right and within a column from top to bottom are mapped to consecutive timesteps until only backbone tasks remain. Each of those is scheduled one at a time.*

2. Sort tasks in LPT order. Let J be the sequence of ordered tasks and L be the sequence of their levels.

3. Select a backbone in the intree and separate free tasks from backbone tasks. Let J_F and J_B be the subsequences of J containing the free respectively the backbone tasks. Let L_F and L_B be the respective subsequences of L.

4. For each free task compute its adjusted level:

$$L'_F[i] := \begin{cases} 1 & \text{if } i = 1, \\ \min(L_F[i], L'_F[i-1]+1) & \text{otherwise.} \end{cases} \tag{1}$$

5. The highest adjusted level of any free task is $h := L'_F[|L'_F|]$. Merge sequences J_F and $J_B[1, ..., h]$ using the adjusted levels from L'_F such that each backbone task is the last in its level. Let J' be the resulting sequence.

6. Let P_1 be the subsequence of tasks at odd indices in J' followed by tasks $J_B[h+1, ..., |J_B|]$. Let P_2 be the subsequence of tasks at even indices in J'.

In the end P_1 and P_2 define the mapping of tasks, where $t_{P_1[i]}$ is scheduled at timestep i on processor 1 and $t_{P_2[i]}$ is scheduled at timestep i on processor 2. Equation (1) formalizes what we called "pulling the strings". Free tasks move to the left as far as possible while maintaining their LPT order and without increasing the number of free tasks on a level beyond 1. This ensures that, after merging backbone tasks into free tasks at their respective levels, no two free tasks with different LPT values are neighbors. In the following Lemma we describe the properties of our algorithm on which we base subsequent proofs.

Lemma 3 *Let $t_1, t_2, ..., t_n$ be the sequence of tasks in J' followed by $J_B[h+1, ..., |J_B|]$. For any tasks t_i, t_j the following holds:*

1. *If t_i and t_j are backbone tasks with $LPT(t_i) < LPT(t_j)$ then $i < j$.*

2. *If t_i and t_j are free tasks with $LPT(t_i) < LPT(t_j)$ then $i < j$.*

3. *If t_i is a free task and t_j a backbone task with $LPT(t_i) \leq LPT(t_j)$ then $i < j$.*

4. *If t_i and t_{i+1} are backbone tasks then all t_j with $i+1 < j \leq n$ are backbone tasks as well.*

5. *If t_i is a free task and t_{i+1} a backbone task with $LPT(t_i) > LPT(t_{i+1})$ then either $i = 1$ or t_{i-1} is a backbone task.*

Proof: Obviously properties 1 and 2 are satisfied, because all tasks are initially sorted in LPT order and the relative order of free tasks and the relative order of backbone tasks is never changed.

In step 5 the first h backbone tasks and all free tasks are merged according to the modified levels of free tasks. Since each backbone task is the last in its level we obtain property 3 for the first h backbone tasks. There are no free tasks on levels greater than h which means that after J' only backbone tasks on higher LPT levels remain. It follows that property 3 holds for the complete sequence.

From equation (1) we know that up to level h there is at least one free task per level. Obviously there is exactly one backbone task per level. It is therefore impossible that two backbone tasks are neighbors in J'. It follows that the first two neighboring backbone tasks are $J'[|J'|]$ and $J_B[h+1]$ which means that property 4 is satisfied.

Let $J'[i]$ be a free task and $J'[i+1]$ a backbone task with $\mathrm{LPT}(J'[i]) > \mathrm{LPT}(J'[i+1])$. Assume $i > 1$. $J'[i]$ must have a modified level that equals $\mathrm{LPT}(J'[i+1])$. Otherwise $J'[i]$ would not be the right neighbor of $J'[i+1]$ after merging. Equation (1) implies that the modified level of $J'[i]$ was determined by the second argument to min. Otherwise the modified level of $J'[i]$ would still be its LPT value. Being determined by the second argument to min in equation (1) ensures that the modified level of $J'[i]$ and $\mathrm{LPT}(J'[i+1])$ is exactly one greater than the modified level of the next free task to the left. Because there is a backbone task on every level and because each backbone task is the last in its level, there must be a backbone task between the free task to the left of $J'[i]$ and $J'[i]$. Obviously $J'[i-1]$ is the only candidate for this backbone task. We conclude that all five properties are satisfied. $\square$

Lemma 4 *Let $t_1, t_2, ..., t_n$ be as in Lemma 3. For any two tasks t_i, t_k with $1 \leq i < k \leq n$ the following holds: if t_i is a free task and t_k is a backbone task with $LPT(t_i) > LPT(t_k)$ then t_{i+1} is a backbone task and either $i = 1$ or t_{i-1} is a backbone task as well.*

Proof: Let t_j be the backbone task closest to t_i with $i < j$. We will show that $j = i+1$. Obviously t_{j-1} is a free task and because free tasks are ordered (property 2) it follows that $\mathrm{LPT}(t_i) \leq \mathrm{LPT}(t_{j-1})$. By property 1 we know that $\mathrm{LPT}(t_j) < \mathrm{LPT}(t_k)$. Therefore $\mathrm{LPT}(t_{j-1}) > \mathrm{LPT}(t_j)$. Property 5 shows that either t_{j-2} is a backbone task or $j-1 = 1$. Because there is no backbone task between t_i and t_j, we conclude that either $i = 1$ or t_{i-1} must be a backbone task and therefore $j = i+1$ and t_{i+1} is a backbone task. $\square$

Lemma 5 *Let $t_1, t_2, ..., t_n$ be as in Lemma 3. For tasks t_i and t_{i+1} with $1 \leq i < n$ the following holds: if t_i or t_{i+1} is a free task then neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$.*

Proof: *case 1:* t_i and t_{i+1} are both free tasks. Assume $\mathrm{LPT}(t_i) \neq \mathrm{LPT}(t_{i+1})$. That means $\mathrm{LPT}(t_i) < \mathrm{LPT}(t_{i+1})$ because free tasks are ordered (property 2). Let t_k be the backbone task on t_i's LPT level. Property 3 implies that $i < k$ and therefore $i+1 < k$. Applying Lemma 4 yields that t_i must be a backbone task which contradicts the case assumption. We conclude that $\mathrm{LPT}(t_i) = \mathrm{LPT}(t_{i+1})$ and by Lemma 1 neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$.

case 2: t_i is a free task and t_{i+1} is a backbone task. Assume $\mathrm{LPT}(t_i) < \mathrm{LPT}(t_{i+1})$. Let t_j be the backbone task on t_i's LPT level. From property 3 we know that $j > i$. Because backbone tasks are ordered (property 1) we have $j < i+1$, a contradiction to $j > i$. Our assumption must have been wrong, therefore $\mathrm{LPT}(t_i) \geq \mathrm{LPT}(t_{i+1})$ and by Lemma 2 neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$.

case 3: t_i is a backbone task and t_{i+1} is a free task. Property 3 shows that $\mathrm{LPT}(t_i) < \mathrm{LPT}(t_{i+1})$. Applying Lemma 2 we get that neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$. $\square$

It is now easy to see that the last step in our algorithm computes a proper schedule. Because neighboring tasks in J' are never constrained by precedence we can map them pairwise from left to right to increasing timesteps. All tasks in $J_B[h+1, ..., |J_B|]$ are backbone tasks so the best we can do is to assign each of them to its own timestep.

Lemma 6 *Let $t_1, t_2, ..., t_n$ be as in Lemma 3, and let $f : T \to \mathbb{N}$ be such that $f(t) = i$ if $t = t_i$. The mapping of tasks defined by P_1 and P_2 in step 6 is equivalent to the following mapping $S : T \to \mathbb{N}$:*

$$S(t) := \begin{cases} \left\lceil \frac{f(t)}{2} \right\rceil & \text{if } f(t) \leq |J'|, \\ \left\lceil \frac{|J'|}{2} \right\rceil + f(t) - |J'| & \text{otherwise.} \end{cases}$$

Proof: Left to the reader. □

Theorem 2 *S is a 2-processor level schedule, and hence optimal.*

Proof: Let $t_1, t_2, ..., t_n$ be as in Lemma 3. We first show that S is a 2-processor schedule. Any two tasks t_i and t_j with $|i-j| \geq 2$ are assigned to different timesteps by S. Therefore no more than 2 tasks are assigned to the same timestep. Lemma 5 implies that two consecutive tasks t_i, t_{i+1} are either two backbone tasks or neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$. The latter ensures that no precedence constraints are violated by S for tasks t_i, t_{i+1} with $i < |J'|$. If t_i and t_{i+1} are both backbone tasks then $i \geq |J'|$ and therefore both tasks are assigned to consecutive timesteps which again makes sure that the precedence constraints are satisfied.

Next we show that S is a level schedule. Timesteps 1 to $\lfloor |J'|/2 \rfloor$ are full and timesteps $\lfloor |J'|/2 \rfloor + 1$ to length(S) are partial. All tasks t_i with $i \geq |J'|$ are backbone tasks because of property 4 and the fact that the last task is always on the backbone by property 3. So there is no free task available at any partial timestep. It follows that S is greedy.

Assume that there exist tasks t_i and t_j with $S(t_i) < S(t_j)$, $\mathrm{LPT}(t_i) > \mathrm{LPT}(t_j)$ and t_j is available at timestep $S(t_i)$. Backbone tasks are ordered (property 1), free tasks are ordered (property 2) and each backbone task t is behind any other task that has an LPT value less or equal to t's LPT value (property 3). It follows that t_i must be a free task and t_j a backbone task. Lemma 4 implies that t_{i+1} is a backbone task and either $i = 1$ or t_{i-1} is a backbone task as well.

case 1: $i = 1$. Then t_i is the first task to be scheduled and t_{i+1} is scheduled at the same timestep as t_i. Obviously t_j can not be available at this timestep because $t_{i+1} \prec t_j$ as both tasks are on the backbone and backbone tasks are ordered.

case 2: t_{i-1} and t_{i+1} are backbone tasks. In this case t_i is scheduled at the same timestep as either t_{i-1} or t_{i+1}. Again t_j is not available at timestep $S(t_i)$ because $t_{i-1} \prec t_{i+1} \prec t_j$.

Both cases contradict our assumption and we conclude that no such tasks t_i and t_j exist, which proves that S is a level schedule. □

3.2 The Parallel Algorithm

In the following we give details on a parallel version of the algorithm using prefix operations, list ranking, and parenthesis matching as basic building blocks. When we visualize intree precedence constraints, the leafs are towards the top and the root is at the bottom of the tree. We assume the precedence tree being represented as a sequence of tasks, each task being associated with a pointer to its successor and its left sibling as is shown in Figure 4. An alternative representation is a sequence of edges where edges that point to the same task are grouped together. It is easy to convert the latter representation into the former and vice versa.

Algorithm TwoProcIntree

(1) Compute the Euler contour path of the precedence tree: We use the Euler tour technique introduced by Tarjan and Vishkin [TV85]. Every tree edge is replaced by two anti-parallel arcs. We build a list of these arcs such that the resulting path runs along the contour of the tree, i.e. the path is an Eulerian tour of the tree. We break this

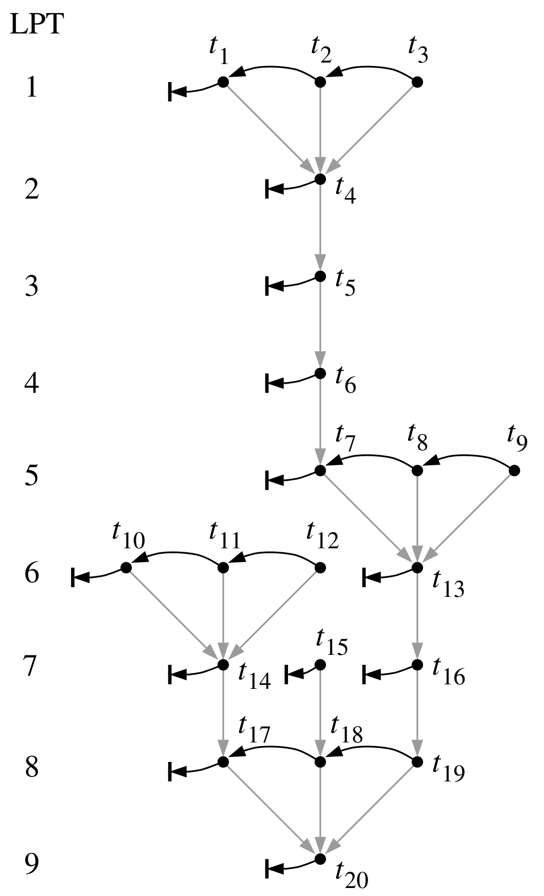

Figure 4: *Input representation of a reduced precedence graph with additional pointers to left siblings.*

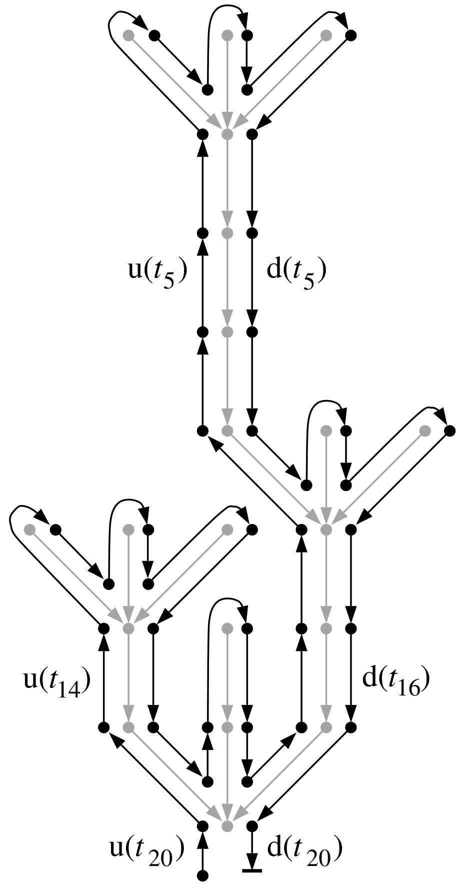

Figure 5: *Euler contour path.*

tour at the root task and obtain a path that corresponds to the order of advancing and retreating along edges during an ordered depth first traversal of the tree.

More formally the Euler tour technique works as follows. For each task we need a pointer to its leftmost predecessor, its successor, and its right sibling. We can easily obtain this data from the given input. For each task t let $\mathrm{d}(t)$ and $\mathrm{u}(t)$ be pointers (initially set to *nil*) representing two anti-parallel arcs that run down[†] respectively up along the tree edge leaving t. For technical reasons we assume that the root task r has $\mathrm{d}(r)$ and $\mathrm{u}(r)$ as well, although no tree edge leaves r. For each task t with no leftmost predecessor (t is then a leaf) set $\mathrm{u}(t)$ to the address of $\mathrm{d}(t)$. Otherwise let t' be its leftmost predecessor and set $\mathrm{u}(t)$ to the address of $\mathrm{u}(t')$. For each task t with no right sibling let t' be its successor and set $\mathrm{d}(t)$ to the address of $\mathrm{d}(t')$. Otherwise let t' be its right sibling and set $\mathrm{d}(t)$ to the address of $\mathrm{u}(t')$ if t' exists.

Obviously the resulting path starts with $\mathrm{u}(r)$ and ends with $\mathrm{d}(r) = nil$. Figure 5 shows the Euler contour path of the tree in Figure 4.

(2) Compute the LPT value of each task: Associate a value of 1 with each upgoing arc and -1 with each downgoing arc in the Euler contour path. Perform list ranking

[†]Note that "down" here means towards the root of the intree (which is at the bottom) and "up" means towards the leaves.

on the Euler contour path while permuting the associated arc values accordingly. Apply prefix sum to the resulting array of arc values. The resulting value of each upgoing arc $u(t)$ is the height of t. Compute the depth of the precedence tree G by applying prefix maximum to the task heights and compute $\mathrm{LPT}(t) = \mathrm{depth}(G) - \mathrm{height}(t) + 1$.

(3) Sort tasks in LPT order: Sorting the vertices of a tree in level order using parenthesis matching and list ranking has been described by Chen and Das in [CD92]. Applied to our context, it works as follows. Associate an opening parenthesis with each downgoing arc and a closing parenthesis with each upgoing arc. Perform list ranking on the Euler path and permute parentheses accordingly. The resulting word is an incomplete parenthesis word. Prepend $\mathrm{depth}(G)$ opening parentheses (imagine the first of these on level 1, the second on level 2, etc.) and append $\mathrm{depth}(G)$ closing parentheses (visualize the first of these on level $\mathrm{depth}(G)$, the second on level $\mathrm{depth}(G) - 1$, etc.) to form a complete parenthesis word. Apply parenthesis matching to this word. For each opening parenthesis we now have a pointer to its matching closing parenthesis. To build a list of all parentheses we associate a pointer with each closing parenthesis as follows. With the i-th of the $\mathrm{depth}(G)$ appended closing parentheses ($i = 2, ..., \mathrm{depth}(G)$) we associate a pointer to the ($\mathrm{depth}(G) + 2 - i$)-th prepended opening parenthesis. The first of the appended parentheses has a *nil* pointer which indicates the end of the list. Every other closing parenthesis b belongs to an upgoing arc $\mathrm{u}(t)$. We associate with b a pointer to the opening parenthesis belonging to $\mathrm{d}(t)$. We now have a list of all parentheses, starting with the first prepended parenthesis and ending with the first appended parenthesis. Apply list ranking to this list of parentheses. Build the subsequence of all closing parentheses that belong to tasks using prefix sum. Replacing each closing parenthesis in this subsequence with its corresponding task yields the sequence of tasks in reverse breadth first order which is LPT order. Let J be the ordered sequence of tasks and L be the sequence of their LPT values.

(4) Select a backbone and separate free tasks from backbone tasks: Associate a value of 1 with each upgoing arc and -1 with each downgoing arc with the exception of the downgoing arc of $J[1]$. Associate -2 with $\mathrm{d}(J[1])$. Task $J[1]$ is the first task of a longest path in G. Perform list ranking on the Euler contour path while permuting the associated arc values accordingly. Apply prefix sum to the resulting array of arc values. Mark tasks whose up- and downgoing arcs carry values which differ by 2. Arc values of tasks not on the selected longest path differ by 1. Divide J into subsequences J_F and J_B of free respectively backbone tasks using prefix sum. Divide L into subsequences L_F and L_B.

(5) Adjust levels of free tasks: For each task $J_F[i]$ compute $\max(i - L_F[i], 0)$ and let Δ be the prefix maximum of the resulting sequence. Then compute $L'_F[i]$ as $i - \Delta[i]$.

(6) Merge backbone tasks and free tasks: Create a sequence B of $\{0, 1\}$ with $B[1] = 0$ and $B[i] = 1$ iff $L'_F[i] \neq L'_F[i-1]$, $1 < i \leq |L'_F|$. Thus a 1 in B indicates a level boundary in L'_F. We want to expand J_F such that whenever there is a level boundary we insert an empty position allocating space for the backbone task at this level. Compute a sequence of new positions for elements in J_F by applying prefix sum to B and adding i to an element at position i. Let P be the resulting sequence. The highest adjusted level of any free task is $h := L'_F[|L'_F|]$. Initialize a sequence J' with $|J_F| + h$ zeroes and store each $J_F[i]$ into $J'[P[i]]$. Create a sequence of $\{0, 1\}$ where the k-th element ($k = 1, ..., |J'|$) is

1 iff $J'[k] = 0$ and let C be the prefix sum of this sequence. Store $J_B[C[k]]$ into $J'[k]$ iff $J'[k]$ is 0. By this we merge backbone tasks $J_B[1, ..., h]$ into J' such that tasks are sorted by level and each backbone task is the last in its level.

(7) Map tasks to processors: Let P_1 be the sequence of tasks at odd positions in J' followed by tasks $J_B[h+1, ..., |J_B|]$. Let P_2 be the sequence of tasks at even positions in J'.

As before P_1 and P_2 define the mapping of tasks to timesteps. Figures 6, 7 and 8 show the intermediate results of algorithm TwoProcIntree when applied to the input from Figure 4. Figure 9 depicts the output sequences P_1 and P_2.

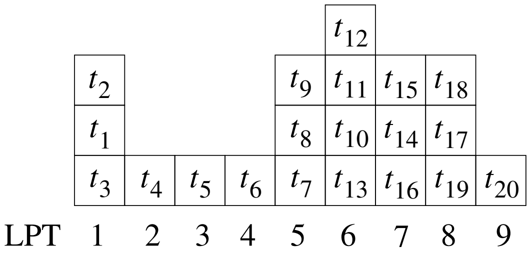

Figure 6: *Tasks from the example sorted by LPT value.*

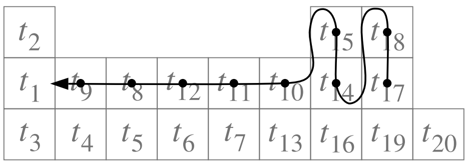

Figure 7: *After adjusting levels of free tasks and merging them with backbone tasks.*

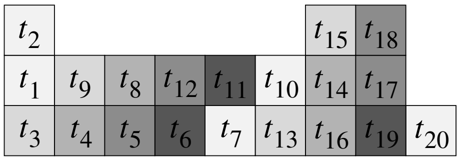

Figure 8: *Consecutive tasks with equal greyscale level are mapped to the same timestep.*

P_2	t_1	t_9	t_8	t_{12}	t_{11}	t_{10}	t_{15}	t_{16}	t_{17}		
P_1	t_2	t_3	t_4	t_5	t_6	t_7	t_{13}	t_{14}	t_{18}	t_{19}	t_{20}
timestep	1	2	3	4	5	6	7	8	9	10	11

Figure 9: *The resulting sequences P_1 and P_2.*

Lemma 7 *The sequence L'_F computed in step (5) of algorithm* TwoProcIntree *satisfies equation (1).*

Proof: For $i = 1$ we get $L'_F[1] = 1 - \Delta[1] = 1 - \max(1 - L_F[1], 0)$. Because $L_F[1] \geq 1$ we have $L'_F[1] = 1$. For $i > 1$ we get

$$\begin{aligned} L'_F[i] &= i - \max_{j=1}^{i}\{\max(j - L_F[j], 0)\} \\ &= i - \max(\max_{j=1}^{i-1}\{\max(j - L_F[j], 0)\}, i - L_F[i]) \\ &= i - \max(i - (L'_F[i-1] + 1), i - L_F[i]). \end{aligned}$$

Assume $L_F[i] < L'_F[i-1] + 1$. It follows that $i - L_F[i] > i - (L'_F[i-1] + 1)$ and therefore the second argument to max dominates. In this case $L'_F[i]$ computes to $L_F[i]$. Otherwise $L_F[i] \geq L'_F[i-1] + 1$. We get $i - L_F[i] \leq i - (L'_F[i-1] + 1)$ which lets the first argument to max dominate and $L'_F[i]$ becomes $L'_F[i-1] + 1$. We conclude that $L'_F[i] = \min(L'_F[i-1] + 1, L_F[i])$. □

Theorem 3 *Let $(T, \prec)$ be a UET 2-processor scheduling problem where the reduced precedence graph $G = (T, E)$ is an intree. Then algorithm* TwoProcIntree *computes a level schedule for $(T, \prec)$.*

Proof: Steps (1) to (4) should be clear. For details on the Euler tour technique we refer to [TV85]. Sorting the vertices of a tree in breadth first order using parenthesis matching and list ranking has been described in [CD92]. From Lemma 7 we know that step (5) in algorithm TwoProcIntree is equivalent to step 4 in the sequential algorithm. It is furthermore obvious that step (6) merges as proposed and step (7) is equivalent to step 6 in the sequential algorithm. By Lemma 6 and Theorem 2 we obtain the desired result. □

4 Greedy Two Processor Schedules for Outtrees

Let $(T, \prec)$ now be a UET 2-processor scheduling problem where the reduced precedence graph $G = (T, E)$ is an outtree. Note that applying algorithm TwoProcIntree to a reversed outtree and reversing the resulting schedule would yield an optimal but not necessarily greedy schedule for the outtree.

Again we select a longest path in G and call tasks on this path backbone tasks. All other tasks in T are called free. We do not produce level schedules this time. Instead we initially sort the outtree tasks by their EPT value which is equal to depth$(G) + 1$ minus the LPT value in the corresponding intree that results from reversing the edges. This way we can reuse part of the intree algorithm when computing greedy optimal schedules for outtrees. The following two observations are equivalent to Lemmas 1 and 2 for the case of outtrees.

Lemma 8 *Let $t, t' \in T$ be on the same EPT level. Then neither $t \prec t'$ nor $t' \prec t$.*

Lemma 9 *For any two tasks $t, t' \in T$ the following holds: if t is a backbone task and t' a free task with EPT$(t) \geq$ EPT(t') then neither $t \prec t'$ nor $t' \prec t$.*

Thus pairing a free task with a backbone task t on a higher EPT level doesn't violate the precedence constraints. We have to find matches for as many backbone tasks as possible giving preference to backbone tasks on lower EPT levels in order to be greedy. We start with an outline of a sequential version of our algorithm.

4.1 A Sequential Algorithm

Assume we have sorted tasks by their EPT values. Figure 10 gives an example where tasks with equal EPT value are grouped into the same column. We order tasks from left to right and within a column from bottom to top. Each backbone task must be the first on its level so we keep them in the bottom row. Figure 10 shows how the free tasks of each block are concatenated and threaded on a string like pearls. This time we pull the strings to the right. But to compute a greedy schedule, we must not pull a string further right if there is no free task to the left of the string that we can move up. This ensures that every EPT level originally occupied by free tasks has at least one free task in the end. In terms of our strings: start pulling with the leftmost string keeping its leftmost task fixed. Pull to the right until either the string is fully stretched or the rightmost task reaches the leftmost task of the next string (or the highest EPT level). In the former case repeat with the next string to the right. In the latter case concatenate the head of the current string with the tail of the next string to the right and proceed by pulling the resulting string. Figure 11 shows the result of pulling strings in the described way. The order of tasks is from left to right and within a single column from bottom to top including backbone tasks. In the next step we insert idle tasks between every two backbone tasks that are adjacent and finally we map tasks at odd positions to processor 1 and tasks at even positions to processor 2 (Figure 12). Let us state the main steps of the algorithm more formally in the following.

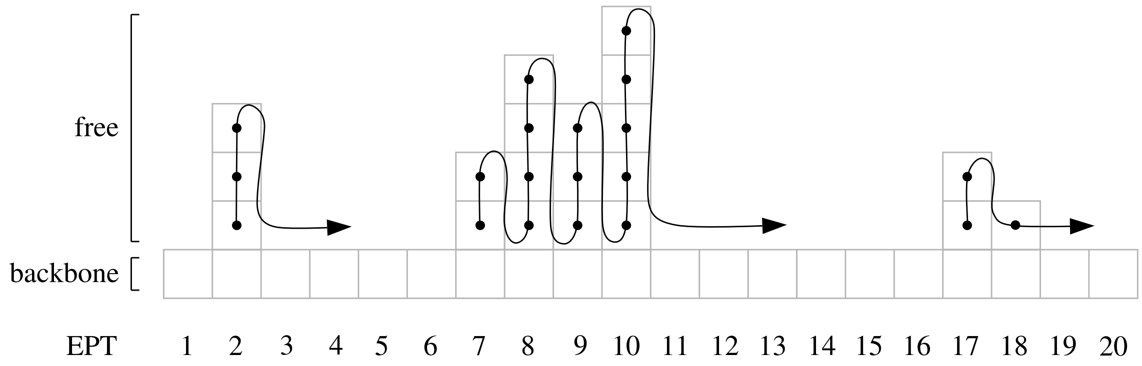

Figure 10: *Tasks ordered by their EPT value. Free tasks in each block are concatenated and threaded on a string.*

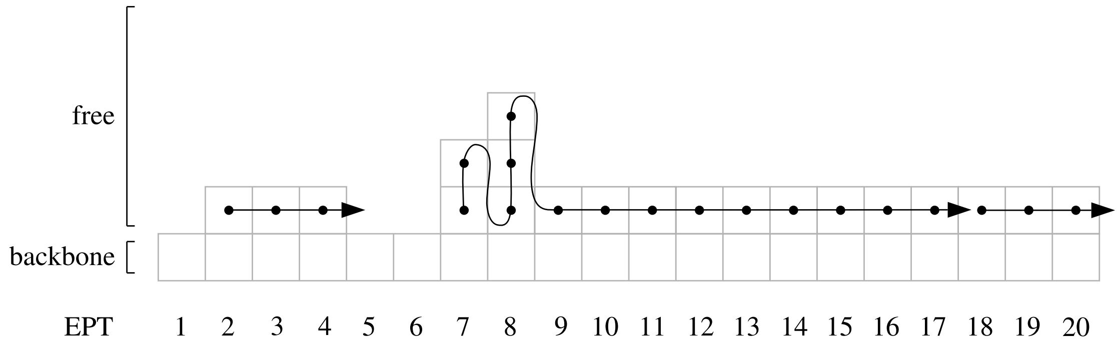

Figure 11: *After pulling strings to the right. We have to ensure that the number of free tasks in each level is not decreased to 0 if it was initially* > 0.

1. Determine the EPT value of each task.

2. Sort tasks in EPT order. Let J be the sequence of ordered tasks and L be the sequence of their EPT values.

3. Select a backbone in the outtree and separate free tasks from backbone tasks. Let J_F and J_B be the subsequences of J containing the free respectively the backbone tasks. Let L_F and L_B be the respective subsequences of L.

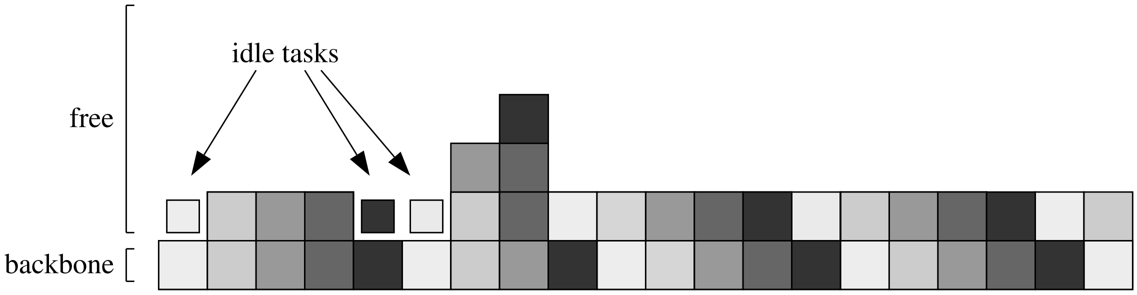

Figure 12: *After inserting idle tasks between adjacent backbone tasks, pairs of tasks from left to right and within a column from bottom to top are mapped to consecutive timesteps.*

4. For each free task compute its minimum level such that there is no more than one free task per level:

$$L_F^1[i] := \begin{cases} L_F[i] & \text{if } i = 1, \\ \max(L_F[i], L_F^1[i-1]+1) & \text{otherwise.} \end{cases} \quad (2)$$

5. For each free task compute its minimum level such that there is at least one free task on each higher EPT level:

$$L_F^2[i] := \begin{cases} \text{depth}(G) & \text{if } i = |L_F|, \\ \max(L_F[i], L_F^2[i+1]-1) & \text{otherwise.} \end{cases} \quad (3)$$

6. Let j be the smallest index such that $L_F^1[j] = L_F^2[j]$ or $j = |L_F|$ if no such index exists. For each free task compute its adjusted level:

$$L'_F[i] := \begin{cases} L_F^1[i] & \text{if } i \leq j, \\ L_F^2[i] & \text{otherwise.} \end{cases} \quad (4)$$

7. Merge sequences J_F and J_B using the adjusted levels from L'_F such that each backbone task is the first in its level. Let J' be the resulting sequence.

8. Insert an idle task between every two adjacent backbone tasks in J'. Let J'' be the resulting sequence.

9. Let P_1 be the subsequence of tasks at odd indices in J'' and P_2 be the subsequence of tasks at even indices.

As before, P_1 and P_2 define the mapping of tasks to timesteps. P_2 may contain idle tasks where $P_2[i] = \mathit{idle}$ means that processor 2 stays idle at timestep i. Note that the length of P_1 may differ from that of P_2 by 1. In this case the missing last task is supposed to be an idle task.

Lemma 10 *Let $t_1, t_2, ..., t_n$ be the sequence of tasks in J'. For any tasks t_i, t_j the following holds:*

1. *If t_i and t_j are backbone tasks with $EPT(t_i) < EPT(t_j)$ then $i < j$.*

2. *If t_i and t_j are free tasks with $EPT(t_i) < EPT(t_j)$ then $i < j$.*

3. *If t_i is a backbone task and t_j a free task with $EPT(t_i) \le EPT(t_j)$ then $i < j$.*

4. *If t_i and t_{i+1} are backbone tasks then there is no $j < i$ such that t_j and t_{j+1} are free tasks.*

5. *If t_i and t_{i+1} are backbone tasks and t_{i+2} is a free task then $EPT(t_{i+1}) = EPT(t_{i+2})$.*

6. *If t_i and t_{i+1} are free tasks then $EPT(t_i) = EPT(t_{i+1})$.*

Proof: Tasks are initially sorted by EPT value and the order of free tasks and the order of backbone tasks is left unchanged throughout the algorithm. It follows that properties 1 and 2 are satisfied. The modified level of a free task is not less than its original EPT value and because step 7 merges each backbone task earlier than all free tasks with a higher modified level, property 3 holds.

Let $L'_F[i] = L'_F[i+1]$ for some $1 \le i < n$. It follows from equations (2), (3) and (4) that $i \ge j$, where j is chosen as in equation (4), because in L^1_F no two positions have equal value. All levels at higher positions in L'_F are therefore determined by L^2_F which means that there is at least one free task per level $\ge L'_F[i]$. It follows that after merging J_F and J_B in step 7 there are no two adjacent backbone tasks in J' behind $J_F[i]$ but $J_F[i]$ and $J_F[i+1]$ are adjacent in J'. Thus property 4 holds.

Let $J'[i]$ and $J'[i+1]$ be backbone tasks and $J'[i+2] = J_F[k]$ be free. It follows that either $k = 1$ or $L'_F[k] - L'_F[k-1] > 1$. Otherwise $J_F[k-1]$ would have been merged between $J'[i]$ and $J'[i+1]$. In both cases the modified level of $J_F[k]$ equals $L^1_F[k]$ because by equation (3) the levels L^2_F don't differ for any adjacent positions by more than 1. The only way in equation (2) that $L^1_F[k] - L^1_F[k-1] > 1$ is that $L^1_F[k] = L_F[k] = \mathrm{EPT}(J_F[k])$. In other words the adjusted level of $J'[i+2]$ is still its original EPT value and therefore $\mathrm{EPT}(J'[i+2]) = \mathrm{EPT}(J'[i+1])$ which means that property 5 holds.

Finally property 6 is satisfied because if two free tasks are adjacent in J' they must have the same adjusted level. This is only possible if their level is determined by L^2_F and if both still have their original levels which means that both are on the same EPT level. □

Lemma 11 *Let $t_1, t_2, ..., t_n$ be as in Lemma 10. For tasks t_i and t_{i+1} with $1 \le i < n$ the following holds: if t_i or t_{i+1} is a free task then neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$.*

Proof: *case 1:* t_i and t_{i+1} are both free tasks. The desired result follows from property 6 and Lemma 8.

case 2: t_i is a free task and t_{i+1} is a backbone task. Property 3 implies that $\mathrm{EPT}(t_i) < \mathrm{EPT}(t_{i+1})$ and by applying Lemma 9 we obtain the desired result.

case 3: t_i is a backbone task and t_{i+1} is a free task. Assume $\mathrm{EPT}(t_i) < \mathrm{EPT}(t_{i+1})$. Let t_j be the backbone task on t_{i+1}'s EPT level. Property 3 implies that $j < i+1$. Because backbone tasks are ordered (property 1) we have $j > i$, a contradiction to $j < i+1$. Our assumption must have been wrong, therefore $\mathrm{EPT}(t_i) \ge \mathrm{EPT}(t_{i+1})$ and by Lemma 9 neither $t_i \prec t_{i+1}$ nor $t_{i+1} \prec t_i$. □

Lemma 12 *Let $t_1, t_2, ..., t_n$ be as in Lemma 10. For tasks t_i and t_{i+1} with $1 \le i < n$ the following holds: if t_i and t_{i+1} are backbone tasks then $t_i \prec t_j$ for all tasks t_j with $j > i$.*

Proof: Obviously the Lemma holds for all backbone tasks t_j with $j > i$. We have to show that it also holds for free tasks t_j, $j > i+1$. Property 5 in Lemma 10 ensures that either there are no free tasks at all behind t_{i+1} or there is a possibly empty sequence of adjacent backbone tasks following t_i ending with backbone tasks t_k, t_{k+1}, $k \geq i$ and free task t_{k+2} with $\text{EPT}(t_{k+1}) = \text{EPT}(t_{k+2})$. No free task t with $\text{EPT}(t) = \text{EPT}(t_k)$ exists, otherwise properties 2 and 3 would ensure that t would fall between t_k and t_{k+2}. Because any free task t_j, $j > k+1$ has an EPT value greater or equal $\text{EPT}(t_{k+1})$ and there is no free task on t_k's EPT level, it follows that $t_k \prec t_j$. Furthermore either $k = i$ or $t_i \prec t_k$, which proves the Lemma. □

Definition 2 *Let $t_1, t_2, ..., t_n$ be as in Lemma 10. The idle task indicator $I : \{1, ..., n\} \rightarrow \{0, 1\}$ is defined as follows:*

$$I(i) := \begin{cases} 1 & \text{if } i > 1 \text{ and } t_{i-1}, t_i \text{ are backbone tasks,} \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 13 *Let $t_1, t_2, ..., t_n$ be as in Lemma 10. If $I(i) = 1$ then $i - 1 + \sum_{j=1}^{i-1} I(j)$ is odd.*

Proof: By induction over i. The first task t_1 is a backbone task (the root of G). By property 3 the second task, if any, is a backbone task as well and therefore $I(1) = 0$ and $I(2) = 1$. Thus the Lemma is true for $i = 1$ and $i = 2$. Assume the Lemma is true up to some $i - 1$. If $I(i) = 1$ then t_{i-1} and t_i are backbone tasks by Definition 2. Let $k < i$ be the greatest index such that $I(k) = 1$. The induction hypothesis implies that $k - 1 + \sum_{j=1}^{k-1} I(j)$ is odd. From t_k up to t_{i-1} are no adjacent free tasks (Property 4 in Lemma 10) and no adjacent backbone tasks (ensured by the choice of k). It follows that the number of tasks between t_k and t_{i-1} is odd and therefore $i - k$ is odd. We have

$$i - 1 + \sum_{j=1}^{i-1} I(j) = i - 1 + \sum_{j=1}^{k} I(j) = \underbrace{k - 1 + \sum_{j=1}^{k-1} I(j)}_{\text{odd}} + \underbrace{I(k)}_{=1} + \underbrace{i - k}_{\text{odd}}$$

which results in an odd value and makes the Lemma true for i. We conclude that the Lemma holds for all $1 \leq i \leq n$. □

Lemma 14 *Let $t_1, t_2, ..., t_n$ be as in Lemma 10 and let $f : T \rightarrow \mathbb{N}$ be such that $f(t) = i$ if $t = t_i$. The mapping of tasks defined by P_1 and P_2 in step 9 is equivalent to the following mapping $S : T \rightarrow \mathbb{N}$:*

$$S(t) := \left\lceil \frac{f(t) + \sum_{j=1}^{f(t)} I(j)}{2} \right\rceil$$

Proof: Left to the reader. □

Theorem 4 *The mapping S from Lemma 14 is a greedy optimal 2-processor schedule.*

Proof: Let $t_1, t_2, ..., t_n$ be as in Lemma 10. We first show that S is a 2-processor schedule. Obviously no more than 2 tasks are assigned to the same timestep. Assume there exist two tasks t_i, t_{i+1} that are constrained by precedence and assigned to the same timestep. Lemma 11 implies that both tasks must be on the backbone. Because $S(t_i) = S(t_{i+1})$ it follows from Lemma 14 that $\sum_{j=1}^{i} I(j) = \sum_{j=1}^{i+1} I(j)$ and therefore $I(i+1) = 0$. This and Definition 2 imply that either t_i or t_{i+1} is a free task, in contradiction to Lemma 11 and our assumption. It follows that S is a 2-processor schedule.

Next we show that S is greedy. Let $S(t_i)$ be a partial timestep. Obviously only one task is available at $S(t_i)$ if $i = 1$ or $i = n$ so we only have to consider the case $1 < i < n$. We will prove by contradiction that t_i and t_{i+1} are backbone tasks. Assume t_i is free. It follows from Definition 2 that $I(i) = 0$ and $I(i+1) = 0$. If we write I for $\sum_{j=1}^{i-1} I(j)$ then

$$S(t_{i-1}) = \left\lceil \frac{i-1+I}{2} \right\rceil, \quad S(t_i) = \left\lceil \frac{i+I}{2} \right\rceil, \quad S(t_{i+1}) = \left\lceil \frac{i+1+I}{2} \right\rceil$$

and therefore either $S(t_{i-1}) = S(t_i)$ or $S(t_i) = S(t_{i+1})$. In both cases $S(t_i)$ can not be a partial timestep. Our assumption was wrong and t_i is a backbone task. Assume next that t_{i+1} is free. It follows from Definition 2 that $I(i+1) = 0$. We obtain

$$S(t_i) = \left\lceil \frac{i+I+I(i)}{2} \right\rceil, \quad S(t_{i+1}) = \left\lceil \frac{i+1+I+I(i)}{2} \right\rceil$$

and because $S(t_i) \neq S(t_{i+1})$ we get that $i + I + I(i)$ must be even. On the other hand we have $S(t_{i-1}) \neq S(t_i)$ from which follows that $I(i) = 1$ and therefore $i - 1 + \sum_{j=1}^{i-1} I(j)$ is odd by Lemma 13. This contradicts $i + I + I(i)$ to be even, because

$$\underbrace{i - 1 + \sum_{j=1}^{i-1} I(j)}_{\text{odd}} = i - 1 + I = \underbrace{i + I + I(i)}_{\text{odd}} - 2.$$

This proves that t_i and t_{i+1} are both backbone tasks. Together with Lemma 12 it follows that $t_i \prec t_j$ for all tasks t_j with $j > i$ and therefore no other task than t_i is available at timestep $S(t_i)$ which settles that S is greedy.

We will now show that S is optimal. Let S' be an optimal 2-processor schedule for $(T, \prec)$. Let $1 \leq i < n$ be the largest index such that t_i and t_{i+1} are backbone tasks. Such an index always exists if $n > 1$, because t_1 and t_2 are backbone tasks. By property 4 there are no adjacent free tasks t_k, t_{k+1} with $k < i$ and because S is greedy, t_i is scheduled at the earliest possible timestep, i.e. $S(t_i) = \mathrm{EPT}(t_i)$. From Lemma 12 we know that there is no task t_j with $j > i$ and $t_i \not\prec t_j$. In other words t_i can not be scheduled earlier in S' than $S(t_i)$ and all tasks t_j, $j > i$ have to be scheduled after $S(t_i)$ in S'. Because S schedules tasks $t_{i+1}, ..., t_n$ in $\lceil (n-i)/2 \rceil$ timesteps and S' can not do better than this, it follows that S' and S are of equal length. □

4.2 The Parallel Algorithm

We assume the precedence outtree being given as a sequence of edges (i, j) where each (i, j) stands for $t_i \prec t_j$. Edges starting at the same task are grouped together.

Algorithm TwoProcOuttree

(1) Reverse precedence tree: For each edge (i, j) associate with t_j a successor pointer to t_i. Let (k, l) be the edge to the left of (i, j) in the input sequence. Associate with t_j a left-sibling pointer to t_l if $i = k$ or *nil* if $i \neq k$. The result is the representation of an intree with each task having a pointer to its successor and to its left sibling.

(2) Compute the Euler contour path of the precedence tree: Apply step (1) of algorithm TwoProcIntree to the reversed precedence tree.

(3) Compute the EPT value of each task: Compute the LPT value of each task in the reversed precedence tree and let $\text{EPT}(t)$ be $\text{depth}(G) + 1 - \text{LPT}(t)$. See step (2) of algorithm TwoProcIntree.

(4) Sort tasks in EPT order: Sort tasks according to their LPT value in the reversed precedence tree and reverse the resulting order. See step (3) of algorithm TwoProcIntree.

(5) Select a backbone and separate free tasks from backbone tasks: Apply step (4) of algorithm TwoProcIntree to the reversed input tree. Let J_F and J_B be the sequences of backbone and free tasks and L_F, L_B be the sequences of their respective EPT value.

(6) Adjust levels of free tasks: For each index in L_F compute $L_F[i] - i$ and let Δ_1 be the prefix maximum of the resulting sequence. Compute $L_F^1[i]$ as $\Delta_1[i] + i$. Let X be the integer sequence $\text{depth}(G) + 1 - |L_F|, \text{depth}(G) + 2 - |L_F|, ..., \text{depth}(G)$. For each index in L_F compute $\max(L_F[i] - X[i], 0)$ and let Δ_2 be the suffix maximum of the resulting sequence. Compute $L_F^2[i]$ as $\Delta_2[i] + X[i]$. Compute the smallest index j such that $L_F^1[j] = L_F^2[j]$, with $j = |L_F|$ if no such index exists, as follows. Create a sequence of length $|L_F|$ with value i at position i iff $L_F^1[i] = L_F^2[i]$ and $|L_F|$ in all other positions. Let R be the prefix minimum of this sequence and set $j := R[|L_F|]$. Let L'_F be the sequence $L_F^1[1, ..., j]$ followed by $L_F^2[j+1, ..., |L_F|]$.

(7) Merge backbone tasks and free tasks: Create a sequence B of length $|L'_F|$ with $B[1] = L'_F[1]$ and $B[i] = L'_F[i] - L'_F[i-1]$ for $1 < i \leq |L'_F|$. Thus B indicates level jumps in L'_F. Apply prefix sum to B and add i to position i. Let P be the resulting sequence. Initialize a sequence J' with n zeroes and store each $J_F[i]$ into $J'[P[i]]$. Create a sequence A of $\{0, 1\}$ where the k-th element ($k = 1, ..., n$) is 1 iff $J'[k] = 0$ and let C be the prefix sum of A. Store $J_B[C[k]]$ into $J'[k]$ iff $J'[k]$ is 0. By this we merge backbone tasks into J' such that tasks are ordered by level and each backbone task is the first in its level.

(8) Map tasks to processors: Let A be from the previous step. Create the idle task indicator I with $I[1] = 0$ and $I[i] = 1$ iff $A[i] = A[i-1] = 1$ and $I[i] = 0$ otherwise for $1 < i \leq n$. Apply prefix sum to I and add i to position i. Let D be the resulting sequence. Initialize a sequence J'' with $D[n]$ zeroes and store $J'[k]$ into $J''[D[k]]$, $1 \leq k \leq n$. Let P_1 be the subsequence of tasks at odd positions in J'' and P_2 be the subsequence of tasks at even positions in J''.

Again P_1 and P_2 define the mapping of tasks to processors. If $P_2[i] = 0$ then processor 2 must stay idle at timestep i. In the following we show that TwoProcOuttree computes greedy optimal schedules.

Lemma 15 *The sequence L'_F computed in step (6) of algorithm* TwoProcOuttree *satisfies equations (2), (3), and (4).*

Proof: We first take a look at the computation of L_F^1 in step (6). For $i = 1$ we have $L_F^1[1] = \Delta_1[1] + 1 = L_F[1]$. For $i > 1$ we get

$$L_F^1[i] \quad = \quad i + \Delta_1[i]$$

$$
\begin{aligned}
&= i + \max_{j=1}^{i}\{L_F[j] - j\} \\
&= i + \max(L_F[i] - i, \max_{j=1}^{i-1}\{L_F[j] - j\}) \\
&= i + \max(L_F[i] - i, L_F^1[i-1] + 1 - i) \\
&= \max(L_F[i], L_F^1[i-1] + 1),
\end{aligned}
$$

which is equivalent to equation (2). Next we analyze L_F^2 in step (6). For $i = |L_F|$ we have

$$
\begin{aligned}
L_F^2[|L_F|] &= \Delta_2[|L_F|] + X[|L_F|] \\
&= \max(L_F[|L_F|] - X[|L_F|], 0) + X[|L_F|] \\
&= X[|L_F|] \\
&= \mathrm{depth}(G),
\end{aligned}
$$

because $L_F[|L_F|] \leq X[|L_F|] = \mathrm{depth}(G)$. For $i < |L_F|$ we get

$$
\begin{aligned}
L_F^2[i] &= \Delta_2[i] + X[i] \\
&= \max_{j=i}^{|L_F|}\{\max(L_F[j] - X[j], 0)\} + X[i] \\
&= \max(\max(L_F[i] - X[i], 0), \max_{j=i+1}^{|L_F|}\{\max(L_F[j] - X[j], 0)\}) + X[i] \\
&= \max(\max(L_F[i] - X[i], 0), L_F^2[i+1] - 1 - X[i]) + X[i] \\
&= \max\{L_F[i], X[i], L_F^2[i+1] - 1\} \\
&= \max(L_F[i], L_F^2[i+1] - 1),
\end{aligned}
$$

because $L_F^2[i] \geq X[i]$ (from equation (3) and the definition of X) which implies $L_F^2[i+1] - 1 \geq X[i+1] - 1 = X[i]$. It follows that L_F^2 computed in step (6) satisfies equation (3). It is furthermore obvious that step (6) computes L'_F from L_F^1 and L_F^2 according to equation (4). □

Theorem 5 *Let $(T, \prec)$ be a UET 2-processor scheduling problem where the reduced precedence graph $G = (T, E)$ is an outtree. Then algorithm* TwoProcOuttree *computes a greedy optimal schedule for $(T, \prec)$.*

Proof: Steps (1) to (5) should be clear. Lemma 15 implies that step (6) computes L'_F according to equations (2), (3), and (4). The reader may easily verify that step (7) merges backbone and free tasks such that each backbone task is the first on its level and that J' is therefore computed as in the sequential algorithm. It is furthermore obvious that step (8) is equivalent to steps 8 and 9 of the sequential algorithm and in conjunction with Lemma 14 and Theorem 4 we conclude that P_1 and P_2 define a greedy optimal schedule. □

Until now we have omitted any details on how to implement the algorithms on a specific machine model. The following section shows that it is easy to find work optimal implementations on the EREW PRAM.

5 PRAM Implementations

The model of computation we consider is the synchronous shared-memory model. It consists of a number of processors each having its own local memory and sharing a common global memory. Processors are controlled by a common clock and in every timestep each of the processors executes one instruction handling a constant number of $\log n$-bit words.

Usually all of them execute the same program on different data. This *data parallelism* is supported by the fact that each processor can check his locally available processor number. We restrict access to global memory in such a way that no two processors can read or write the same memory cell at the same timestep. A machine with these properties is called an exclusive-read, exclusive-write parallel random access machine, or EREW PRAM for short. For details and variations of this model we refer to [FW78].

Theorem 6 *There are EREW PRAM implementations of algorithms* TwoProcIntree *and* TwoProcOuttree *that run in $O(\log n)$ time using $n/\log n$ processors, where n is the number of tasks to be scheduled.*

Proof: Algorithms TwoProcIntree and TwoProcOuttree apply basic parallel functions a constant number of times. We have to show that all those functions can be computed within the desired resource bounds. It is obvious that we can construct the Euler contour path of an intree in constant parallel time on n processors if for each vertex a pointer to its successor and its left sibling is given. If we dedicate each processor to $\log n$ tasks we can compute the Euler path in $O(\log n)$ time on $n/\log n$ processors. A technique similar to the Euler path technique was used by Wyllie [Wyl79] for tree traversals. As a general technique it was introduced by Tarjan and Vishkin [TV85]. A prefix sum algorithm for the EREW PRAM running in $O(\log n)$ time using $n/\log n$ processors was developed by Ladner and Fischer [LF80]. For optimal list ranking on the EREW PRAM we can either use the algorithm of Cole and Vishkin [CV88] or the one developed by Anderson and Miller [AM88]. Algorithms for parenthesis matching running in $O(\log n)$ time using $n/\log n$ processors of an EREW PRAM can be found in [AMW89], [TLC89], or [CD91]. $\square$

6 Conclusion and Open Problems

We have analyzed the problem of scheduling n tasks constrained by a tree-like precedence relation on 2 processors. Two work optimal EREW PRAM algorithms running in $O(\log n)$ time using $n/\log n$ processors have been presented, one computing greedy optimal schedules for intrees, the other for outtrees. Compared to the fastest existing parallel algorithm for intrees [HM87a] the number of operations is reduced by a factor of $O(n^2 \log n)$, in the case of outtrees [DUW86] the factor is $O(n \log n)$.

Future research has to show whether the techniques used in this paper can successfully be applied to constant $m > 2$. Furthermore it remains an open problem if there is a parallel algorithm that schedules UET trees with m part of the problem instance running in $O(\log n)$ time with less than $O(n^2 \log n)$ operations.

Another possible line of research is to find efficient implementations of our algorithms on networks such as the hypercube. It is easy to replace the basic parallel PRAM functions we use by their respective hypercube counterparts and thereby obtain hypercube algorithms. For prefix operations, concentration routing, and parenthesis matching there are hypercube implementations known that require only logarithmic time [Sch80, NS81, MW92]. Unfortunately our algorithms apply list ranking and permutation routing as well and the fastest known list ranking algorithm requires time $O(\log^2 n \log\log\log n \log^* n)$ on n hypercube processors [HM93] and the best known deterministic routing algorithm is Sharesort [CP90] which requires time $O(\log n(\log\log n)^2)$. If the precedence tree is given as a parenthesis word (which is essentially the Euler contour path) no list ranking is necessary but still tasks must be sorted in LPT order. Thus the resulting algorithms run in time $O(\log n(\log\log n)^2)$ on n hypercube processors. It would be worth investigating whether one can find modifications to our algorithms that result in faster implementations on the hypercube.

References

[AM88] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *Proceedings of the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures, AWOC 88 (Corfu, Greece, June/July 1988)*, LNCS 319, pages 81–90, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1988. Springer-Verlag.

[AMW89] Richard J. Anderson, Ernst W. Mayr, and Manfred K. Warmuth. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, September 1989.

[BGJ77] P. Brucker, M. Garey, and D. S. Johnson. Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness. *Math. Oper. - Res.*, 2:275–284, 1977.

[BHS+94] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21:4–14, 1994.

[Bru82] John Bruno. Deterministic and stochastic scheduling problems with treelike precedence constraints. In M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling, Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems (Durham, England, July 1981)*, volume C84 of *NATO Advanced Study Institutes Series*, pages 367–374. D. Reidel Publishing Company, 1982.

[CD91] Calvin C.-Y. Chen and Sajal K. Das. A cost-optimal parallel algorithm for the parentheses matching problem on an EREW PRAM. In V. K. Prasanna Kumar, editor, *Proceedings of the Fifth International Parallel Processing Symposium (Anaheim, California, April/May 1991)*, pages 132–137, Washington, D.C., 1991. IEEE Computer Society Press.

[CD92] Calvin C.-Y. Chen and Sajal K. Das. Breadth-first traversal of trees and integer sorting in parallel. *Inf. Process. Lett.*, 41:39–49, 1992.

[CG72] E. G. Coffman, Jr. and R. L. Graham. Optimal scheduling for two processor systems. *Acta Inf.*, 1:200–213, 1972.

[CP90] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (Baltimore, Maryland, May 1990)*, pages 193–203, New York, 1990. ACM SIGACT, ACM Press.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling, Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17(1):128–142, February 1988.

[DS84] Eliezer Dekel and Sartaj Sahni. A parallel matching algorithm for convex bipartite graphs and applications to scheduling. *J. Parallel Distrib. Comput.*, 1:185–205, 1984.

[DUW86] Danny Dolev, Eli Upfal, and Manfred K. Warmuth. The parallel complexity of scheduling with precedence constraints. *J. Parallel Distrib. Comput.*, 3:553–576, 1986.

[FKN69] M. Fujii, T. Kasami, and K. Ninamiya. Optimal sequencing of two equivalent processors. *SIAM J. Appl. Math.*, 17(4):784–789, July 1969.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA)*, pages 114–118, New York, 1978. ACM, ACM Press.

[Gab82] H. N. Gabow. An almost-linear algorithm for two-processor scheduling. *J. ACM*, 29(3):766–780, 1982.

[HM87a] David Helmbold and Ernst W. Mayr. Fast scheduling algorithms on parallel computers. In Franco P. Preparata, editor, *Advances in Computing Research; Parallel and Distributed Computing*, volume 4, pages 39–68. JAI Press Inc., Greenwich, CT-London, 1987.

[HM87b] David Helmbold and Ernst W. Mayr. Two processor scheduling is in $\mathcal{NC}$. *SIAM J. Comput.*, 16(4):747–759, August 1987.

[HM93] Volker Heun and Ernst W. Mayr. A new efficient algorithm for embedding an arbitrary binary tree into its optimal hypercube. Technical Report TUM-I9321, Institut für Informatik, Technische Universität München, August 1993.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[May81] Ernst W. Mayr. Well structured parallel programs are not easier to schedule. Technical Report STAN-CS-81-880, Computer Science Dept., Stanford University, September 1981.

[MW92] Ernst W. Mayr and Ralph Werchner. Optimal routing of parentheses on the hypercube. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '92 (San Diego, California, June 29 – July 1, 1992)*, pages 109–117, New York, 1992. ACM SIGACT, ACM SIGARCH, ACM Press.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Trans. Comput.*, C-30(2), 1981.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, October 1980.

[TLC89] W. W. Tsang, T. W. Lam, and F. Y. L. Chin. An optimal EREW parallel algorithm for parenthesis matching. In Fred Ris and Peter M. Kogge, editors, *Proceedings of the 1989 International Conference on Parallel Processing, Vol. 3 (Penn State University, August 1989)*, pages 185–192, University Park-London, 1989. Pennsylvania State University Press.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, November 1985.

[Ull75] J. D. Ullman. $\mathcal{NP}$-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

Appendix

What follows is an implementation of algorithm TwoProcIntree written in Nesl, a strongly typed, applicative, data-parallel language. Nesl supplies nested parallelism through the ability to apply any function concurrently over each element of a sequence, even if the function is itself parallel and the elements of the sequence are themselves sequences. For details we refer to [BHS+94]. In the sequel we assume the reader to be familiar with basic concepts of Nesl. Our implementation is non-optimal for two reasons:

- We use pointer jumping to rank the elements of a list and a standard sorting algorithm to sort tasks in level order. Both need $O(n \log n)$ operations. Work optimal algorithms for list ranking or parenthesis matching are non-trivial and would have lengthened the appendix unnecessarily.
- Nesl supplies the programmer with a care-free environment where all low-level details such as allocating data to processors get handled dynamically. This costs an extra $O(\log n)$ overhead factor on the EREW PRAM.

Functions `prefix_sum` and `prefix_max` are derived from Nesls built-in scan operations. `repermute` is the inverse function of the built-in `permute` operation. `level`, `task`, and `tasks` handle access to pairs of integers, where a task's level is the first component and a task's number the second. Note that Nesl-sequences start with index 0 and $\#s$ denotes the length of a sequence s.

```
function prefix_sum(s) =
  let r = take(rotate(plus_scan(s), -1), #s - 1)
  in r ++ [r[#r - 1] + s[#s - 1]];

function prefix_max(s) =
  let r = take(rotate(max_scan(s), -1), #s - 1)
  in r ++ [max(r[#r - 1],s[#s - 1])];

function repermute(x, permutation) =
  let index = plus_scan(dist(1, #x));
  in permute(x, permute(index, permutation));

function level(p)    = let (x,y) = p; in x;
function task(p)     = let (x,y) = p; in y;
function tasks(s)    = let (x,y) = unzip(s); in y;
function last_elt(s) = s[#s-1];
```

```
function jump_pointer(pointer, distance, count) =
  let
    nc = count - 1;
    np = {if x == -1 then x else pointer[x]: x in pointer};
    nd = {if x == -1 then y else y + distance[x]: y in distance; x in pointer}
  in
    if (nc > 0)
    then jump_pointer(np, nd, nc)
    else nd;

function rank_list(pointer) =
  let
    distance = {if x == -1 then 0 else 1: x in pointer};
    count = ceil(log(float(#pointer), 2.0));
  in
    jump_pointer(pointer, distance, count);

function euler_path(successor, left_sibling) =
  let
    index = plus_scan(dist(1, #successor));
    pred_insert ={(successor[i], i): i in index |
                  successor[i] >= 0 and left_sibling[i] == -1};
    leftmost_predecessor = dist(-1, #successor) <- pred_insert;
    sibling_insert = {(left_sibling[i],i): i in index | left_sibling[i] >= 0};
    right_sibling = dist(-1, #successor) <- sibling_insert;
    up = {if lp == -1 then 2*i + 1 else 2*lp:
          i in index; lp in leftmost_predecessor};
    down = {if rs == -1 then 2*s + 1 else 2*rs:
            rs in right_sibling; s in successor};
  in
    interleave(up, down);

function levels(euler_permutation) =
  let
    edge_values = interleave(dist(-1, #euler_permutation/2),
                             dist(1, #euler_permutation/2));
    permuted_edges = prefix_sum(permute(edge_values, euler_permutation));
  in
    odd_elts(repermute(permuted_edges, euler_permutation));

function backbone(first_backbone_task, euler_permutation) =
  let
    edge_values = interleave(dist(-1, #euler_permutation/2),
                             dist(1, #euler_permutation/2));
    edge_values = edge_values <- [(first_backbone_task*2 , -2)];
    permuted_edges = prefix_sum(permute(edge_values, euler_permutation));
    edge_values = repermute(permuted_edges, euler_permutation);
    up_edges = odd_elts(edge_values);
    down_edges = even_elts(edge_values);
  in
    {if diff(u,d) == 2 then 1 else 0: u in up_edges; d in down_edges};

function move(free_tasks) =
  let
    (lpt_levels, tasks) = unzip(free_tasks);
    index  = prefix_sum(dist(1, #free_tasks));
    offset = prefix_max({max(idx-lpt,0): idx in index; lpt in lpt_levels});
  in
    zip({idx-off: idx in index; off in offset}, tasks);

function merge(backbone_tasks, free_tasks) =
  let
    index = plus_scan(dist(1, #free_tasks));
    end_of_level = {if level(x) /= level(y) then 1 else 0:
                    x in free_tasks; y in rotate(free_tasks, -1)};
    offset = plus_scan(end_of_level);
```

```
    newpos = {idx + off: idx in index; off in offset};
    expanded = dist((0,0), newpos[#newpos-1]+2) <- zip(newpos, free_tasks);
    backbone_pos = plus_scan({if level(x)==0 then 1 else 0: x in expanded});
    schedule = {if level(ft) == 0 then backbone_tasks[pos] else ft:
                ft in expanded; pos in backbone_pos};
    p1 = even_elts(schedule)
         ++ drop(backbone_tasks,level(last_elt(free_tasks)));
    p2 = odd_elts(schedule);
  in
    (tasks(p1), tasks(p2));

function two_proc_intree(taskid, successor, left_sibling) =
  let
    index = plus_scan(dist(1, #taskid));
    euler_permutation = rank_list(euler_path(successor, left_sibling));
    vertex_levels = levels(euler_permutation);
    level_order = rank(vertex_levels);
    highest_level = last_elt(permute(vertex_levels, level_order));
    lpt_levels = {highest_level - level + 1: level in vertex_levels};
    lpt_order = {#level_order-1-pos: pos in level_order};
    first_backbone_task = permute(index, lpt_order)[0];
    bb_flags = backbone(first_backbone_task, euler_permutation);
    tasks = permute(zip(lpt_levels, taskid), lpt_order);
    bb_flags = permute(bb_flags, lpt_order);
    bb_tasks = {x: x in tasks; bb_task in bb_flags | bb_task == 1};
    free_tasks = {x: x in tasks; bb_task in bb_flags | bb_task == 0};
  in
    merge(bb_tasks, move(free_tasks));
```

Running the functions defined above against the example from Figure 4 yields:

```
<Nesl> two_proc_intree(
>[ 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20],
>[ 3, 3, 3, 4, 5, 6,12,12,12,13,13,13,15,16,17,18,19,19,19,-1],
>[-1, 0, 1,-1,-1,-1,-1, 6, 7,-1, 9,10,-1,-1,-1,-1,-1,16,17,-1]);
Compiling..Writing..Loading..Running..Exiting..Reading..
it = ([2, 3, 4, 5, 6, 7, 13, 14, 18, 19, 20],
[1, 9, 8, 12, 11, 10, 15, 16, 17]) : [int], [int]
```