Fakultät für Informatik
der Technischen Universität München

Work Efficient Parallel Scheduling Algorithms

Hans Stadtherr

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. E. Jessen
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. E. W. Mayr
	2.	Univ.-Prof. Dr. Dr. h.c. W. Brauer

Die Dissertation wurde am 21. Oktober 1997 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12. März 1998 angenommen.

Abstract

Scheduling the execution of parallel algorithms on parallel computers is a main issue in current research. Parallel computers can be used to solve scheduling problems very fast and we might be able to tackle new applications, where schedules must be obtained very quickly. Although the importance of parallel scheduling algorithms has been widely recognized, only few results have been obtained so far. In this thesis, we present new and efficient parallel scheduling algorithms.

A classical problem in scheduling theory is to compute a minimal length schedule for executing n unit length tasks on m identical parallel processors constrained by a precedence relation. This problem is $\mathcal{NP}$-complete in general, but there exist a number of restricted variants of it that have polynomial solutions and are still of major interest.

First, we show that if the precedence constraints are restricted to be trees, then an optimal schedule can be computed in time $O(\log n \log m)$ using $n / \log n$ processors of an EREW PRAM. Hence, for those cases where m is not part of the input but a constant, our algorithm is work and time optimal. Second, we present a parallel algorithm that optimally schedules arbitrary precedence constraints on two processors. It runs in time $O(\log^2 n)$ and uses $n^3 / \log n$ processors. In addition, we show that optimal two processor schedules can be computed much more efficiently, if the precedence constraints are restricted to be series parallel graphs. In this case, an optimal schedule can be computed in logarithmic time and a linear number of operations suffice.

Finally, we investigate the parallel complexity of scheduling with unit interprocessor communication delays. In this extended setting the schedule additionally takes into account the time required to communicate data between processors. After finishing a task, one unit of time must pass before any of its successors can be started on a different processor, in order to allow for transportation of results from one task to the other. The problem of computing minimal length schedules in this setting is $\mathcal{NP}$-complete, even if precedence constraints are restricted to be trees. The only nontrivial class of precedence constraints for which polynomial solutions are known, is the class of interval orders. We present a parallel algorithm that solves the problem for interval orders in time $O(\log^2 n)$ using $n^3/\log n$ processors. This is the first $\mathcal{NC}$ algorithm for this problem.

Acknowledgments

I would like to take this opportunity to thank all people who contributed to this work.

Special thanks go to my advisor, Ernst W. Mayr, for introducing me to parallel scheduling algorithms. Without his guidance and the insights he gave me, this thesis would be a different one.

For providing a supportive and most enjoyable environment in which to work, I thank my colleagues Ernst Bayer, Stefan Bischof, Thomas Erlebach, Tom Friedetzky, Volker Heun, Ulla Koppenhagen, Klaus Kühnle, Angelika Steger, and Ulrich Voll. I am grateful to Stefan Bischof and Jacques Verriet for their helpful comments on an earlier version of Chapter 7.

Very special thanks go to Andrea, for carefully reading most of this thesis, but above all, for her love and understanding. Finally, I wish to thank my parents and my sister for all their support.

Contents

CHAPTER 1
Introduction

A traditional field of application for scheduling theory are industrial manufacturing processes. Over the past decade, increased interest in scheduling problems originated from a different area. The availability of parallel computers and high performance communication networks gave rise to new directions in scheduling research. For one thing, new types of scheduling problems need to be solved in order to efficiently utilize parallel computers, *e.g.*, scheduling parallel computations with interprocessor communication delays. Second, parallel computers themselves can be used to obtain schedules much faster than previously possible on sequential machines. Using parallel scheduling algorithms we might be able to tackle new applications where schedules must be obtained very quickly. Although the importance of parallel scheduling algorithms has been widely recognized, only few results have been obtained so far. This is mainly due to the fact that many of the techniques used in sequential scheduling algorithms are $\mathcal{P}$-complete, *i.e.*, they presumably can not be parallelized. In order to obtain fast parallel scheduling algorithms, new techniques have to be developed.

In this thesis, we present parallel algorithms that solve fundamental scheduling problems relevant to parallel and networked computing. The algorithms are exponentially faster than their sequential counterparts. However, our main focus is on work efficiency, *i.e.*, we are interested in parallel algorithms that perform as few operations as possible in order to achieve exponential speedup.

In the following section we introduce the multiprocessor scheduling problem and variants thereof. The introduction is accompanied by an overview of known results on the subject. Thereafter, an outline of this thesis is given.

1.1 Scheduling Parallel Computations

A parallel computation consists of a number of tasks $t_1, \ldots, t_n$ that have to be executed by a number of parallel processors $P_1, \ldots, P_m$. We assume that all processors are identical and all tasks are known in advance. Let us first consider the case where all tasks can be executed independently from each other. Task t_j requires processing time p_j and is to be processed sequentially by exactly one processor. A *schedule* is an assignment of tasks to processors. The *load* of a processor is the sum of the processing times of the tasks assigned to it, and the *length* of a schedule is the maximum load of any processor. We wish to find a schedule of minimal length. This problem can also be phrased as a decision problem, where we ask whether there exists a schedule of length at most k. Unfortunately, even when $m = 2$, the decision problem is $\mathcal{NP}$-hard, as was shown by Karp [Kar72]. It is therefore highly unlikely that we can find a polynomial algorithm that computes schedules of minimal length.

Graham was one of the first who studied approximation algorithms for the multiprocessor scheduling problem. He proved that if tasks are put into a list sorted in nonincreasing order of processing times, and whenever a processor becomes idle, it executes the next task in the list, then the length of the schedule produced is at most $\frac{4}{3} - \frac{1}{3m}$ times the optimum [Gra69]. Later, several researchers improved on this result, and finally, it was discovered that the problem possesses a polynomial approximation scheme, *i.e.*, for every $\varepsilon > 0$, there exists an $(1+\varepsilon)$-approximation algorithm that runs in polynomial time [HS85, May85]. The running time of these algorithms grows exponentially in $\frac{1}{\varepsilon^2}$, and as a result by Garey and Johnson [GJ78] shows, this can not be improved significantly, since the problem is $\mathcal{NP}$-hard in the strong sense.

1.1.1 Scheduling with Precedence Constraints

Let us now extend our model by precedence constraints. A partial order $\prec$ on the set of tasks is given and task t_i must be finished before t_j starts if $t_i \prec t_j$. Now, a schedule assigns tasks to processors *and* time intervals. Again, we wish to find a schedule of minimal length. This problem is $\mathcal{NP}$-hard, even if all tasks have identical execution times [Ull75]. Graham has shown that if we put the tasks into a list in arbitrary order, and whenever a processor becomes idle, it executes the leftmost unscheduled task in the list that is ready for execution, then the length of the produced schedule is at most $2 - \frac{1}{m}$ times the optimum [Gra69]. Unfortunately, there is no hope for a polynomial approximation scheme. For the case of unit processing time, to decide whether a schedule of length at most 3 exists is $\mathcal{NP}$-complete [LRK78]. As a consequence, no polynomial approximation algorithm with performance bound better than $\frac{4}{3}$ can exist unless $\mathcal{P} = \mathcal{NP}$. It is interesting to note that no approximation algorithm is known that significantly improves on Graham's $2 - \frac{1}{m}$ performance bound, even when all processing times are equal.

In the following we focus on the case where all tasks have equal processing times. As already mentioned, the problem is still $\mathcal{NP}$-hard in this case, but if we put further restrictions on the problem, several interesting results can be obtained. An early result by Hu [Hu61] shows that if the precedence constraints are restricted to be trees, then a schedule of minimal length can be computed in polynomial time. Later, Brucker, Garey, and Johnson gave an implementation of Hu's algorithm that runs in linear time [BGJ77]. The algorithm is based on a simple strategy: put the tasks into a list in nonincreasing order of the height each task has in the tree. Whenever a processor becomes idle, it executes the leftmost unscheduled task in the list that is ready for execution. Several researchers have also addressed the parallel complexity of scheduling trees. Fast parallel algorithms have been proposed by Helmbold and Mayr [HM87a], and Dolev, Upfal, and Warmuth [DUW86]. The most efficient of them runs in time $O(\log^2 n)$ if $n / \log n$ processors are used.

Another type of precedence constraints that have been analyzed in the literature are interval orders. Interval orders are those partial orders that can be defined using intervals on the real line. With each task a closed interval on the real line is given and task t_i must be finished before t_j starts if the interval of t_i is completely to the left of the interval of t_j. Again, a quite simple strategy yields optimal schedules. Let the successor set of a task t_i be the set of tasks that can not start before t_i is finished. Papadimitriou and Yannakakis [PY79] showed that if tasks are put into a list sorted by nonincreasing size of successor sets, and whenever a processor becomes idle, it executes the leftmost unscheduled task in the list that is ready for execution, then one obtains a schedule of minimal length. If the precedence constraints are given as a precedence graph, then the schedule can be computed in time $O(n + e)$, where e is the number of edges in the graph. Sunder and He developed a parallel algorithm for this problem [SH93] that runs in polylogarithmic time and requires n^4 processors. Subsequently, Mayr gave an improved parallel algorithm that requires only n^3 processors [May96].

Instead of restricting the precedence constraints to be of a certain type, one might hope to obtain polynomial solutions for the case when the number m of processors is fixed and not part of the problem instance. For any fixed $m > 2$, the complexity of the problem is still unknown, but for the case $m = 2$ polynomial solutions have been found. Fujii, Kasami, and Ninomiya proposed the first polynomial algorithm based on the idea to construct an optimal two processor schedule from a maximum matching in the incomparability graph of the given partial order [FKN69]. Subsequently, faster algorithms were given by Coffman and Graham, and by Sethi [CG72, Set76]. Finally, Gabow [Gab82] proposed an algorithm that requires time linear in the size of the precedence graph when combined with a result on static union-find given in [GT85].

The parallel complexity of the two processor scheduling problem was first investigated by Vazirani and Vazirani [VV85]. They gave an algorithm based on a randomized algorithm for maximum matchings with an expected running time that is polylogarithmic. The first efficient deterministic parallel (*i.e.*, $\mathcal{NC}$) algorithm was developed by Helmbold and Mayr [HM87b]. Their algorithm runs in time $O(\log^2 n)$ and requires n^{10} processors. Since then, a number of attempts have been made to develop more efficient parallel algorithms [MJ89, JSS91, Jun92]. The only known provably correct algorithm that is more efficient than the algorithm of Helmbold and Mayr still requires n^5 processors and is a combination of results given in [HM87b] and [Jun92].

Another class of precedence constraints that has been considered in the past are series parallel orders. Series parallel orders are defined recursively as follows. A task system is series parallel if it either consists of only one task or can be split into two series parallel task systems T_1 and T_2 such that either all tasks of T_1 must be finished before any task in T_2 can start or each task of T_1 can be executed independently of each task in T_2. This class of precedence constraints is quite rich, in particular, all parallel programs that follow the divide and conquer paradigm have series parallel task systems. In general, series parallel orders are not “easier” to schedule than unrestricted precedence constraints: The problem of computing minimal length schedules for tasks with unit processing times is still $\mathcal{NP}$-complete [May81, War81]. Moreover, it is unknown whether polynomial algorithms exist that compute optimal m-processor schedules for series parallel orders if m is a constant greater than two, which is the same situation as with unrestricted precedence constraints.

1.1.2 Scheduling with Communication Delays

In the past few years, scheduling with communication delays has received considerable attention. In this extended setting the schedule additionally takes into account the time required to communicate data between processors. More precisely, for each pair of tasks with $t_i \prec t_j$, a delay c_{ij} is given and after finishing a task t_i, time c_{ij} must pass before t_j can be started on a different processor, in order to allow for transportation of results from one task to the other. If the successor task is executed on the same processor, then no delay is needed unless it is necessary to wait for results from some other processor. Rayward-Smith was one of the first who considered communication delays. For the case where all task processing times and all communication delays are equal, he proved that the problem of computing a minimal length schedule is $\mathcal{NP}$-hard, and he showed that any greedy schedule has length at most $3 - \frac{2}{m}$ times the optimum [RS87]. A more involved algorithm was presented by Hanen and Munier [HM95]. It is based on an LP relaxation and achieves a performance bound of $\frac{7}{3} - \frac{4}{3m}$. This bound is achieved even for the case where processing times and communication delays are not equal, as long as the largest communication time is shorter than the shortest processing time of a task. Another interesting result has recently been obtained by Möhring, Schäffter, and Schulz. For the case of unit processing times and unit communication times, they gave a $(2 - \frac{1}{m})$-approximation algorithm for series parallel orders [MSS96].

There is also a result known concerning nonapproximability. For unit processing times and unit communication times, Hoogeveen, Lenstra, and Veltman have shown that it is $\mathcal{NP}$-hard to decide whether a schedule of length at most 4 exists [HLV92]. This implies that, unless $\mathcal{P} = \mathcal{NP}$, no polynomial approximation algorithm can achieve a performance bound better than $\frac{5}{4}$.

Let us now focus on the case where all processing times and all communication delays are equal. Surprisingly, if we restrict the precedence constraints to be trees, the problem remains $\mathcal{NP}$-hard, as was shown by Lenstra, Veldhorst, and Veltman [LVV93]. On the other hand, if the number m of processors is fixed and not part of the problem instance, then optimal schedules for trees can be computed in polynomial time [VRKL96]. For the case $m = 2$, linear time algorithms for scheduling trees are known [LVV93, GT93, VRKL96]. Recently, an $O(n^2)$ time algorithm has been proposed that optimally schedules series parallel orders on two processors [FLMB96]. The complexity of scheduling arbitrary precedence constraints on two processors is still unknown.

Another class of precedence constraints that has been considered in the context of unit processing time and unit communication delays are interval orders. Picouleau [Pic92] has shown that the same strategy that was used by Papadimitriou and Yannakakis for scheduling interval orders without communication delays can also be applied here, *i.e.*, a list schedule computed from a list of all tasks sorted by nonincreasing size of successor sets has minimal length. A similar algorithm has been proposed by Ali and El-Rewini [AER95]. The running time of their algorithm is $O(nm + e)$, where e denotes the number of edges in the given precedence graph. It is interesting to note that, other than interval orders, no nontrivial class of precedence constraints is known that allows computation of optimal schedules in polynomial time if the number m of processors is part of the input.

A variant of the scheduling problem with communication delays is to allow task duplication. It might be profitable to execute the same task more than once. Instead of waiting for some task t_i to be finished and to transport the results from the processor P that executes t_i to another processor P', it might be better to schedule a duplicate of t_i on processor P'. Unfortunately, the result obtained in [HLV92] implies that even with task duplication (and unit processing times and unit communication times) the problem remains $\mathcal{NP}$-hard. In the case of outtree precedence constraints, task duplication can decrease the length of a schedule. In the case of intrees, it doesn't help at all. A result regarding approximability has been obtained by Hanen and Munier. In [HM97], a $(2-\frac{1}{m})$-approximation algorithm is given for scheduling tasks with unit processing time and unit communication delays. One should keep in mind that task duplication is only applicable if tasks do not interact with the environment. It might be impossible to execute a task more than once.

1.2 Preview

Our work on parallel scheduling algorithms begins in the next chapter with some notation and concepts related to asymptotic complexity, graphs, partial orders, and parallel algorithms. We continue in Chapter 3 with an introduction into basic parallel techniques and algorithms. With one exception, these algorithms can either be found in the literature or result from straightforward applications of known algorithms. Some of them have been adapted to our specific needs.

In Chapter 4 we turn to the problem of scheduling unit execution time tasks with tree precedence constraints. We propose a strategy where the given tree is partitioned into $m+1$ task sets, and to each of the m available processors one of these task sets is assigned as a basic load. The remaining subset of tasks is used to balance the load between processors. We show that if the tree is carefully partitioned and the balancing is properly done, then optimal schedules are obtained. An efficient parallel implementation is given that outperforms existing parallel algorithms. In particular, if m is fixed and not part of the problem instance, then our algorithm is work and time optimal on the EREW PRAM, *i.e.*, it runs in $O(\log n)$ time and performs $O(n)$ operations.

Chapter 5 is devoted to the two processor scheduling problem. Helmbold and Mayr proposed an interesting strategy for this problem [HM87b]. In order to learn about the structure of an optimal schedule (without actually computing one) the algorithm determines for each pair of tasks t and t' the length of an optimal two processor schedule for the tasks that are at the same time successors of t and predecessors of t'. These "scheduling distances" are then used to construct the actual schedule. The scheduling distance algorithm required n^5 processors, and to compute the final schedule, n^{10} processors were necessary. Since then some improvements have been achieved. In [Jun92], it was shown how the final schedule can be constructed using only n^3 processors, but the resource bounds for the scheduling distance computation have not been improved. In Chapter 5, we present an algorithm that computes scheduling distances in $O(\log^2 n)$ time on $n^3/\log n$ processors. We furthermore show that an optimal schedule can be obtained within the same bounds. We close the chapter by applying this result to the maximum matching problem in co-comparability graphs. We show that, using our two processor scheduling algorithm as a subroutine, maximum matchings in co-comparability graphs can be computed in $O(\log^2 n)$ time on n^3 processors.

We stay with the two processor scheduling problem for one more chapter. In Chapter 6, we show that if the precedence constraints are restricted to be series parallel orders, then an optimal two processor schedule can be obtained much more efficiently than in the general case. In fact, our algorithm for this problem runs in logarithmic time and performs a linear number of operations.

In contrast to the previous chapters, Chapter 7 deals with communication delays. We present two results related to interval orders. First, we improve on the result given by Ali and El-Rewini [AER95] and show that optimal schedules for interval orders can be computed by a sequential algorithm in time $O(n+e)$, where e is the number of edges in the given precedence graph. Second, we present a parallel algorithm that runs in logarithmic time if n^3 processors are employed. The parallel algorithm proceeds in two stages. In stage one, we determine for every task the number of timesteps required to schedule all of its predecessors in the interval order. In stage two, we use these numbers to construct an instance of a different scheduling problem where tasks are not constrained by a precedence relation but have individual release times and deadlines. We compute an optimal schedule for this instance, and thus obtain an optimal schedule for the original problem. Our algorithm is the first $\mathcal{NC}$ algorithm for scheduling with communication delays.

In Chapter 8, we close with concluding remarks and some open problems.

CHAPTER 2

Preliminaries

The purpose of this chapter is to introduce basic definitions and notation. We start with some mathematical preliminaries concerning asymptotic complexity, Landau symbols, and logarithms. Thereafter, we review graph theoretic concepts. We continue with sections on partial orders and precedence constraints. In the last section of this chapter we introduce the model of computation that our parallel scheduling algorithms are based upon.

2.1 Asymptotic Complexity

We measure the bounds on the resources (for example, time and work) required by our programs as a function of the input size. Hereby, we are interested in the *worst-case* complexity: A resource bound states the maximum amount of that resource our program requires on any input of size n. These bounds are expressed *asymptotically* using the following notation. For functions f and g, we say f is $O(g)$ (or $f = O(g)$) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$. We say f is $\Omega(g)$ (or $f = \Omega(g)$) if there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$, for all $n \geq n_0$. If f is $O(g)$ and f is $\Omega(g)$, then we say f is $\Theta(g)$.

Instead of $(f(n))^k$ we write $f^k(n)$, while $f^{(k)}(n)$ is used to denote the value of f applied k times to n, *i.e.*, $f^{(1)}(n) = f(n)$ and $f^{(k+1)}(n) = f^{(k)}(f(n))$. For instance, $\log^2(n) = (\log(n))^2$ but $\log^{(2)}(n) = \log(\log(n))$. Usually, we write $\log n$ instead of $\log(n)$ and unless stated otherwise, we assume $\log n$ to be base 2. By $\log^* n$ we denote the smallest integer k such that $\log^{(k)} n \leq 1$. For a real number x, let $\lceil x \rceil$ denote the smallest integer greater or equal to x and let $\lfloor x \rfloor$ denote the greatest integer less than or equal to x.

A function f is said to be *polynomial* if $f(n) = n^{O(1)}$, it is *linear* if $f(n) = O(n)$, it is *polylog* or *polylogarithmic* if $f(n) = \log^{O(1)} n$, and it is *logarithmic* if $f(n) = O(\log n)$.

2.2 Graphs

In this section we review some graph theoretic concepts. A *multigraph* $G = (V, E)$ consists of a finite set of vertices V and a finite multiset of edges E. Each edge is a pair (v, w) of distinct vertices (*i.e.*, $v \neq w$). If E is a set, then G is a *graph*. If the edges of G are unordered pairs, then G is an *undirected multigraph* (*undirected graph*). Otherwise the edges are ordered pairs, and G is a *directed multigraph* (*directed graph*). Directed multigraphs (graphs) are also called *multidigraphs* (*digraphs*). To emphasize that an edge is undirected, we sometimes write $\{v, w\}$ instead of (v, w). The terms defined in the following for graphs can also be applied to multigraphs.

If (v, w) is an edge, then v and w are *adjacent* to each other and the edge (v, w) is *incident* to v and w. We also say that (v, w) *connects* v and w, and v and w are the *endpoints* of (v, w). A directed edge (v, w) *leaves* v, *enters* w, is an *outgoing* edge of v, and an *incoming* edge of w.

The *outdegree* of a vertex v in a digraph is the number of outgoing edges of v. The *indegree* of v is the number of incoming edges of v. The *degree* of a vertex v in an undirected graph is the number of edges incident to v. A vertex whose indegree equals zero is called a *source*, and a vertex whose outdegree equals zero is called a *sink*. A vertex that has no adjacent vertices is *isolated*.

A *path* of *length* k is a sequence of vertices $v_0, \ldots, v_k$ such that (v_i, v_{i+1}) is an edge for $0 \leq i < k$. Vertices v_0 and v_k are the *endpoints* of the path and all other vertices are *internal vertices*. A path *visits* or *contains* vertices $v_0, \ldots, v_k$ and edges (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k), and it *avoids* all other vertices and edges. A path is *simple* if no internal vertex is visited twice. Unless stated otherwise, we assume that paths are simple. If $v_0 = v_k$ and $k \geq 2$, then the path is a *cycle*. A graph is *acyclic* if it contains no cycles.

A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. G' is a *proper subgraph* of G, if G' is a subgraph of G and $V' \subset V$ or $E' \subset E$. Graph G' is an *induced subgraph* of G if, in addition, all edges of G with both endpoints in V' are contained in E'. In this case, we say G' is the subgraph of G induced by V'.

A graph $\overline{G} = (V, \overline{E})$ is the *complement* of the graph $G = (V, E)$ if for every pair of distinct vertices v and w, the edge (v, w) is in $\overline{E}$ iff $(v, w) \notin E$.

Let G be an undirected graph. A subgraph G' of G is a *clique* if each pair of vertices in G' is adjacent. A clique G' is a *maximum clique* if no other clique of G contains more vertices. A subset $V' \subseteq V$ of vertices is called an *independent set*, if no two vertices in V' are connected. A *k-coloring* for a graph $G = (V, E)$ is a partition of the vertex set $V = X_1 \uplus X_2 \uplus \cdots \uplus X_k$ such that each subset X_i is an independent set. The elements of X_i are said to be "colored" with color i. As a consequence, every vertex is colored with exactly one color and two connected vertices are colored differently. If there exists a *k-coloring* for G, then G is said to be *k-colorable*. The *chromatic number* of G is the smallest possible k for which G is k-colorable.

A cycle of length 3 is called a *triangle*. An edge e is a *chord* of a cycle c, if the endpoints of e are contained in c but e is not. A chord e of a cycle c is a *triangular chord* of c, if there exists a triangle such that two of the three edges of the triangle belong to c and the other edge of the triangle is e. An undirected graph is called *chordal* if every cycle of length four or greater has a chord. Equivalently, an undirected graph is chordal iff every cycle either is a triangle or contains a triangular chord.

A graph is *connected* if for every pair of vertices $v \neq w$, there exists a path from v to w or a path from w to v. A *connected component* of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G.

A subset M of edges of an undirected graph G is called a *matching* in G if no two edges of M share the same endpoint. M is called a *maximum matching* if no other matching in G contains more edges than M.

A directed acyclic graph (*dag*) $G = (V, E)$ is called *transitive* or *transitively closed* if for every pair of distinct vertices v and w such that there is a path from v to w, the edge (v, w) exists. The *transitive closure* $G^+ = (V, E^+)$ of G is the dag such that (v, w) is an edge in G^+ iff $v \neq w$ and there is a path from v to w in G. An edge (v, w) is *redundant* if there exists a path from v to w that avoids (v, w). A dag with no redundant edges is *transitively reduced*. The *transitive reduction* $G^- = (V, E^-)$ of G is the unique transitively reduced dag that has the same transitive closure as G.

Let v be a vertex in a dag. The *depth* of v, denoted by *depth*(v), is the length of a longest path from any source to v plus 1. In particular, all sources have depth 1. The *height* of v, denoted by *height*(v), is the length of a longest path from v to any sink plus 1. In particular, all sinks have height 1. The height of a dag G, denoted by *height*(G), is the maximum height of any vertex in G.

The vertices of a dag can be partitioned into *levels*. We say vertex v is on level ℓ, and write $\mathit{level}(v) = \ell$, if the height of v is ℓ. We adopt the convention to draw directed acyclic graphs with edges pointing downwards such that all vertices on the same level have the same vertical coordinate. Consequently, if $\ell > \ell'$ we say that level ℓ is *higher* than ℓ' and ℓ' is *lower* than ℓ.

Let v and w be vertices of a dag. We say that v is a *predecessor* of w (and w is a *successor* of v) if there is a path from v to w. We say that v is an *immediate predecessor* of w (and w is an *immediate successor* of v) if (v,w) is an edge and v,w is the only path from v to w.

2.2.1 Bipartite Graphs

An undirected graph $G = (V,E)$ is called *bipartite* if its vertex set can be partitioned into two disjoint independent sets A and B. Equivalently, G is bipartite iff it is 2-colorable. If the partition of the vertex set is clear from the context, then we denote the bipartite graph G by the triple (A,B,E). A bipartite graph is *convex* on A if the vertices of A can be ordered, say $A = \{a_1,\ldots,a_m\}$, such that for every vertex $b \in B$, the subset of vertices in A connected to b forms an interval in the order of A, *i.e.*, if $\{a_i,b\} \in E$ and $\{a_{i+k},b\} \in E$, then $\{a_{i+r},b\} \in E$ for all r between 0 and k. For every $b \in B$, let $\mathit{begin}(b)$ denote the number of the first vertex in A connected to b and let $\mathit{end}(b)$ denote the number of the last vertex in A connected to b. A convex bipartite graph can concisely be represented by a list of tuples $(\mathit{begin}(b_1),\mathit{end}(b_1))$, ..., $(\mathit{begin}(b_n),\mathit{end}(b_n))$, where $B = \{b_1,\ldots,b_n\}$. For instance, the tuple representation of the convex bipartite graph depicted in Figure 2.1 is $(1,2)$, $(2,5)$, $(1,6)$, $(5,6)$.

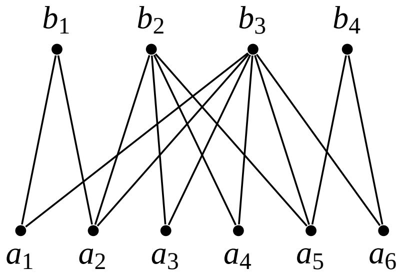


Figure 2.1: *A convex bipartite graph.*

2.2.2 Trees

An undirected graph G is a *tree* if it is connected and acyclic. A graph G is a *forest of trees* if every connected component of G is a tree. A *rooted tree* is a tree $T = (V,E)$ with one designated vertex *root*(T), called the *root*. In the sequel, we only consider rooted trees and every tree is understood to be a rooted tree. Vertices of degree 1, other than the root, are called *leaves*. All vertices that are not leaves are *internal vertices*. Note that between two vertices v and w of a tree there exists exactly one path.

Let $\{u,w\}$ be an edge in a tree T. Then u is the *parent* of w and w is a *child* of u if the path from the root of T to u does not contain w. If v and w have the same parent, then v is a *sibling* of w. In drawings of undirected trees, the vertices are placed such that each vertex is above its children. In particular, the root is at the top. Vertex u is an *ancestor* of v and v is a *descendant* of u if the path from the root to v contains u. Note that every vertex is ancestor and descendant of itself. The *subtree* of T rooted at vertex u is the subgraph of T induced by the vertices that are descendants of u. A vertex u is *common ancestor* of vertices v and w if u is ancestor of v and ancestor of w. The *lowest common ancestor* of two vertices v and w is the root of the smallest subtree that contains v and w.

The *depth* of a tree vertex v, denoted by *depth*(v), is the length of the path from the root to v plus 1. In particular, the depth of the root is 1. The depth of a tree T, denoted by *depth*(T), is the maximum depth of any vertex in T. The *height* of a vertex v, denoted by *height*(v), is the length of a longest path from v to any leaf plus 1. In particular, the height of a leaf is 1. The height of a tree T, denoted by *height*(T), is the height of its root.

A tree is *ordered* if the children of each vertex are ordered. The order on siblings is given by specifying the *left sibling* and the *right sibling* of each vertex. The first child of a vertex has no left sibling, and the last child has no right sibling.

A *traversal* of a tree is a list of its vertices that contains each vertex exactly once. The *depth-first traversal* of an ordered tree T consists of the root of T followed by the depth-first traversals of its subtrees from left to right. The *preorder number* of vertex v, denoted by *pre*(v), is i if v is the i-th vertex in the depth-first traversal of T. The *breadth-first traversal* of an ordered tree T consists of tasks sorted by depth such that the vertices that have equal depth are sorted by their preorder number. An example for a depth-first traversal and a breadth-first traversal is given in Figure 2.2.

A tree is called *binary* if every vertex has at most two children. A binary

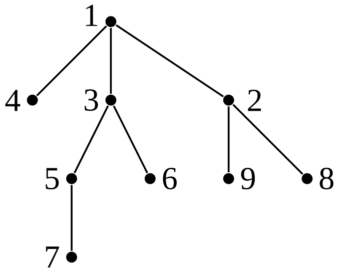


Figure 2.2: *An ordered tree. Its depth-first traversal is* $1,4,3,5,7,6,2,9,8$. *For instance, the preorder number of vertex 4 is 2, and the preorder number of vertex 9 is 8. The breadth-first traversal of this tree is* $1,4,3,2,5,6,9,8,7$.

tree is *proper* if every internal vertex has exactly two children. A binary tree is *complete* if it is proper and all leaves have the same depth.

2.2.3 Inforests and Outforests

A dag is called an *inforest* if the outdegree of every vertex equals either zero or one. In an inforest, a vertex with an outdegree equal to zero is called a *root*. An inforest is an *intree* if only one root is present (cf. Figure 2.3b). Similarly, a dag is called an *outforest* if the indegree of every vertex equals either zero or one. Here, all vertices with an indegree equal to zero are called roots. An outforest with only one root is an *outtree* (cf. Figure 2.3a).

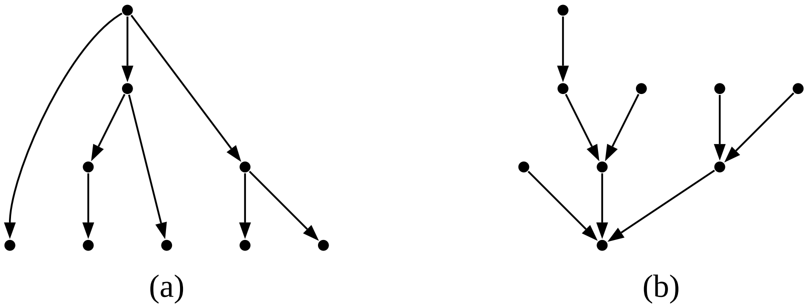


Figure 2.3: *(a) An outtree and (b) an intree.*

Note that intrees and outtrees become trees if the direction of edges is ignored. We make this distinction between undirected trees and directed trees because we use them in two different contexts. Directed trees occur as precedence graphs in scheduling problems, while trees are used as computational structures in our algorithms. In the former case the direction of edges is vital, while in the latter case the direction is unimportant.

2.2.4 Vertex Series Parallel Digraphs

A digraph is a *vertex series parallel digraph* (VSP digraph) if it can be constructed by applying recursively the following rules:

1. A digraph consisting of a single vertex is a VSP digraph.
2. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are VSP digraphs, then
   (a) the *series composition* $G_1 \circ G_2 := (V_1 \cup V_2, E_1 \cup E_2 \cup (O_1 \times I_2))$ is a VSP digraph, where O_1 is the set of sinks of G_1 and I_2 is the set of sources of G_2, and
   (b) the *parallel composition* $G_1 \cup G_2 := (V_1 \cup V_2, E_1 \cup E_2)$ is a VSP digraph.

Note that every VSP digraph is acylic. A VSP dag G can be represented by a decomposition tree that reflects the recursive structure of G. An ordered proper binary tree T is a *decomposition tree* for G if each leaf of T corresponds to a vertex of G, and vice versa. Each internal vertex v of T is either marked with "P" or "S", representing a series or parallel composition of the VSP dags that correspond to the subtrees of T rooted at the left child and the right child of v. We adopt the convention that an "S" vertex joins the sinks of the VSP dag of its left subtree with sources of the VSP dag of its right subtree. Figure 2.4 shows a VSP dag together with a decomposition tree for it.

Let v be a vertex in the decomposition tree T for a VSP dag G. Then $G(v)$ denotes the VSP dag defined by the subtree of T rooted at v. In particular, $G(root(T)) = G$. For each vertex v, $G(v)$ is called a *defining subgraph of G with regard to T*. For instance, let v denote the lowest common ancestor of vertex 1 and vertex 5 in Figure 2.4b. Then $G(v)$ is the subgraph of the VSP dag in Figure 2.4a induced by the vertices $1, 2, 3, 5$.

Let G be a VSP dag and let T be a decomposition tree for it. The *breadth-first traversal of G with regard to T* is the list of all vertices of G sorted by depth such that the vertices that have equal depth in G are sorted by their preorder number in T. For instance, the breadth-first traversal of the VSP dag in Figure 2.4a with regard to the decomposition tree in Figure 2.4b is $1, 2, 4, 3, 5, 7, 6$.

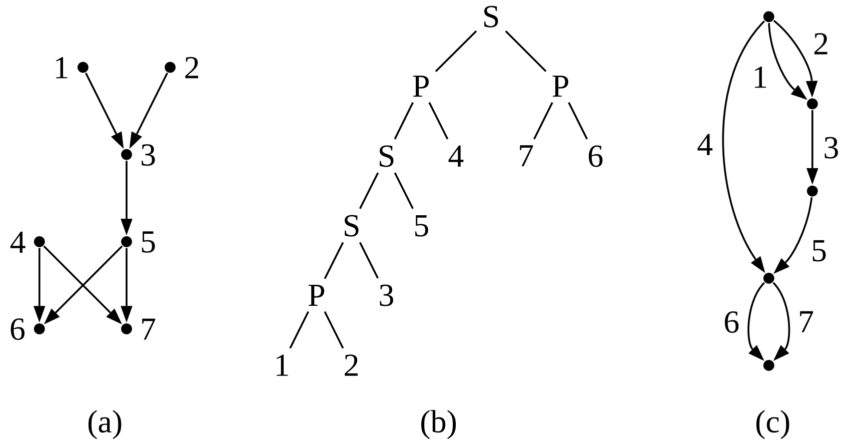


Figure 2.4: *(a) A vertex series parallel dag, (b) a decomposition tree for it, and (c) its line digraph inverse (see Subsection 2.2.5).*

2.2.5 Edge Series Parallel Multidigraphs

A class of graphs that is closely related to VSP dags is the class of edge series parallel multidigraphs. A multidigraph is an *edge series parallel (ESP) multidigraph* if it can be constructed by recursively applying the following rules:

1. A digraph consisting of two vertices connected by a single edge is an ESP multidigraph.

2. If G_1 and G_2 are ESP multidigraphs, then the multidigraphs constructed by each of the following operations are ESP:

   (a) *two terminal series composition*: identify the sink of G_1 with the source of G_2.

   (b) *two terminal parallel composition*: identify the source of G_1 with the source of G_2 and identify the sink of G_1 with the sink of G_2.

Note that ESP multidigraphs are acyclic. ESP multidags are also called *two terminal series parallel multidags* because every ESP multidag has exactly one source and one sink.

An ESP multidag G can be represented by a decomposition tree. An ordered proper binary tree T is a *decomposition tree* for G, if each leaf of T corresponds to an edge of G, and vice versa. Each internal vertex v of T is either marked with “P” or “S”, representing a series or parallel composition of the ESP multidags that correspond to the subtrees of T rooted at the left child and the right child of v. We adopt the convention that an “S” vertex identifies the sink of the ESP multidag of its left subtree with the source of the ESP multidag of its right subtree.

Obviously, ESP multidags and VSP dags are related. To capture this relationship formally, we require the notion of line digraphs. The *line digraph* of a multidigraph $G = (V,E)$ is the digraph $L(G) = (V',E')$ such that there exists a bijective mapping f from E to V' and $(f(e_1), f(e_2))$ is an edge in E' iff e_1 and e_2 are edges in E with $e_1 = (u,v)$ and $e_2 = (v,w)$.

Lemma 2.1 ([VTL82]) *A multidigraph with one source and one sink is ESP iff its line digraph is a VSP dag.*

Clearly, every multidigraph has a unique line digraph. Hence, the VSP dag that is the line digraph of an ESP multidag is uniquely determined. The “inverse” multidigraph of a line digraph is not uniquely determined in general, but one can show the following [HN60]: For every line digraph there exists an inverse multidigraph that has at most one source and at most one sink (take an arbitrary inverse multidigraph and identify its sources with each other and identify its sinks with each other). If we only consider multidigraphs with at most one sink and at most one source as inverses of line digraphs, then every line digraph G has a unique inverse graph, which we denote by $L^{-1}(G)$. $L^{-1}(G)$ is called the *line digraph inverse* of G.

This fact can easily be seen in the case of ESP multidags: For every VSP dag G there exists exactly one ESP multidag $L^{-1}(G)$ such that $L(L^{-1}(G)) = G$. For instance, the ESP multidag in Figure 2.4c is the line digraph inverse of the VSP dag in Figure 2.4a. Note that the tree in Figure 2.4b is not only a decomposition tree for the VSP dag, but it is also a decomposition tree for its line digraph inverse. Each leaf of the tree represents a vertex of the VSP dag and an edge of its line digraph inverse. This fact holds in general, and the converse is also true, *i.e.*, a decomposition tree for an ESP multidag is a decomposition tree for its line digraph.

2.3 Partial Orders

Let X be some set. A *binary relation* R on X is a subset of $X \times X$. We write xRy if $(x,y) \in R$, while $x\not{R}y$ means $(x,y) \notin R$. A binary relation is said to be

- *irreflexive* if $x\not{R}x$ for all $x \in X$,

- *asymmetric* if xRy implies $y\not{R}x$ for all $x,y \in X$, and
- *transitive* if xRy and yRz implies xRz for all $x,y,z \in X$.

A binary relation on X is a *partial order* if it is irreflexive and transitive. Note that every partial order is asymmetric. Let R be a partial order on X. The *comparability graph* of R is the undirected graph $G = (X,E)$ with $\{x,y\} \in E$ iff xRy or yRx. The *incomparability graph* of R is the complement of its comparability graph.

There is a one-to-one correspondence between partial orders and transitively closed dags. The dag that *corresponds* to the partial order R on X is $G_R = (X,E)$ with $(x,y) \in E$ iff xRy, for all $x,y \in X$. A partial order R is a *tree order* if the transitive reduction of G_R is either an inforest or an outforest. R is a *series parallel order* if the transitive reduction of G_R is a vertex series parallel dag.

2.3.1 Interval Orders

A partial order R on X is an *interval order* if, for all $x,y,v,w \in X$, xRy and vRw implies that xRw or vRy. Interval orders can also be characterized using intervals on the real line. A partial order R on X is an interval order iff there exists a mapping I from X to closed intervals on the real line such that $I(x)$ is completely to the left of $I(y)$ iff xRy. The mapping I is called an *interval representation* of the interval order R. A graph that is the incomparability graph of an interval order is called an *interval graph*. As is well known, a partial order is an interval order iff its incomparability graph is chordal [GH64].

2.4 Precedence Constraints

Task systems with *precedence constraints* consist of a set of tasks T and a partial order $\prec$ on T. If $x \prec y$ then we say that y is a *successor* of x or y *succeeds* x and x is a *predecessor* of y or x *precedes* y. If x is a predecessor of y and there exists no task that is both successor of x and predecessor of y then x is an *immediate predecessor* of y and y is an *immediate successor* of x. In every schedule for $(T,\prec)$, a task x must be finished before y starts if y is a successor of x.

It is convenient to represent the partial order $\prec$ by a directed acyclic graph. A *precedence graph* for $(T,\prec)$ is a directed acyclic graph G with vertex set T such that for all $x,y \in T$ there is a directed path from x to y iff $x \prec y$. Since tasks are also vertices of the given precedence graph, the terms for vertices of a graph can equally well be applied to tasks. For instance, each task has a depth, a height, and a level.

2.5 Parallel Algorithms

We describe our parallel algorithms on a high level and mostly in terms of basic parallel functions that are generally accepted as basic building blocks for parallel algorithms (see Chapter 3). This enables us to concentrate on the algorithmic idea required to solve a scheduling problem and to abstract from features specific to certain parallel machine architectures. Efficient implementations of these basic functions are available on a number of parallel architectures (cf. [JáJ92, Lei92]).

2.5.1 Computational Model

Nevertheless, in order to analyze resource requirements and to be able to compare our results with existing work, we assume as a model of computation the *parallel random access machine* (PRAM) [FW78], a model that is widely used as a platform for describing parallel algorithms. The PRAM 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 $O(\log n)$-bit words. Communication between processors is carried out by exchanging data through the shared memory.

Different variants of the PRAM model have been considered with respect to the constraints on simultaneous access to the same memory location. In the *exclusive read exclusive write* (EREW) PRAM, each memory location can be accessed by at most one processor at a time. In the *concurrent read exclusive write* (CREW) PRAM, each memory cell can be written to by at most one processor at a time. If no limitations are posed on memory accesses, then the PRAM is of the *concurrent read concurrent write* (CRCW) type. The CRCW PRAM model can be differentiated further by how concurrent writes are handled. The *common* CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The *arbitrary* CRCW PRAM allows an arbitrary processor to succeed, and the *priority* CRCW PRAM assumes that the indices of the processors are linearly ordered, and allows the one with minimum index to succeed. All variants of the PRAM have been shown to be equally powerful except for logarithmic factors in running time or processor requirements.

The advantage of PRAMs over parallel machine models with distributed memory is that in the PRAM setting, algorithms are easier to describe and easier to analyze. The drawback of PRAMs is the somewhat “unrealistic” assumption of a globally shared memory. It has been shown, however, that PRAMs can be simulated on distributed memory machines with only a small increase in running time [AHMP87, KLM92].

2.5.2 Performance of Parallel Algorithms

The *worst-case cost* of a parallel algorithm is given by three functions $T(n)$, $P(n)$, and $W(n)$. Function $T(n)$ is the maximum number of timesteps the algorithm requires to process an input of size n, $P(n)$ is the number of processors that must be employed in order to achieve the time bound $T(n)$, and $W(n)$ is the maximum number of operations that are performed on an input of size n. The work $W(n)$ does not necessarily equal $T(n) \cdot P(n)$, since not all processors may be active in each timestep.

A parallel algorithm is said to be *work optimal* if it is proved that no other parallel algorithm can exist that solves the same problem using asymptotically less operations. A parallel algorithm is *time optimal* if it is proved that no other parallel algorithm can be asymptotically faster. Both terms are used with regard to a specific machine architecture. For instance, an algorithm may be time optimal with regard to the EREW PRAM, despite the fact that a faster algorithm exists for the CRCW PRAM.

2.5.3 Complexity Classes

Obviously some computational problems are easier to solve than others. A main issue in computer science is to provide a mathematical framework in which problems can be classified with regard to their complexity. It is generally accepted to say that a problem can be solved efficiently if it can be solved in polynomial time. For parallel algorithms, a similar notion has been introduced. A problem is said to be efficiently solvable in parallel if it can be solved in polylogarithmic time using a polynomial number of processors. This class of problems is commonly referred to as the class $\mathcal{NC}$. In the sequel we assume that the reader is familiar with basic terms of complexity theory such as $\mathcal{NP}$-completeness and $\mathcal{P}$-completeness. For an introduction to $\mathcal{NP}$-completeness theory we refer to the book by Garey and Johnson [GJ79]. A detailed treatment of the limitations of parallel computation can be found in the book by Greenlaw, Hoover, and Ruzzo [GHR95].

CHAPTER 3

Basic Parallel Techniques

In this chapter we present basic parallel techniques and algorithms that we use in our parallel scheduling algorithms. Most of these algorithms have been taken from the literature. Some of them have been adapted to meet specific requirements of our scheduling algorithms, and others are straightforward applications of known algorithms. The only "basic" algorithm that can not be found elsewhere is the one that computes a breadth-first traversal of a series parallel graph (Section 3.11).

3.1 Rescheduling and Brent's Scheduling Principle

Suppose we are given a PRAM algorithm with time bound $T(n)$ that requires $P(n)$ processors. If only $p \leq P(n)$ processors are available on a specific machine, we have to *reschedule* the algorithm. We partition the $P(n)$ processors into p groups of size at most $\lceil P(n)/p \rceil$ and let each of the p available processors simulate the processors of a distinct group. Hence, each parallel step of the algorithm can be simulated by the p processors in time $\lceil P(n)/p \rceil$. As a consequence, the time required to execute the algorithm on p processors is $T(n) \cdot \lceil P(n)/p \rceil$. In the sequel, we will use rescheduling quite often and usually without further notice.

The concept of rescheduling can be extended in the following way. Suppose we are given an algorithm that runs in $T(n)$ parallel steps and performs $W_i(n)$ operations in step i, $1 \leq i \leq T(n)$. We simulate each set of $W_i(n)$ operations by p processors in $\lceil W_i(n)/p \rceil$ parallel steps. Then the time spent is $\sum_i \lceil W_i(n)/p \rceil \leq \sum_i (W_i(n)/p+1) = W(n)/p+T(n)$, where $W(n)$ is the total number of operations performed by the given algorithm. This scheduling principle is also known as *Brent's scheduling principle*. Brent used this technique to evaluate arithmetic expressions efficiently in parallel [Bre74].

Brent's scheduling principle has to be applied carefully. We have to make sure that the assignment of work to processors that is necessary to simulate a step can be done. More precisely, for each parallel step, each of the p processors must know whether it is active or not, and if it is active, it must know the instruction it has to perform and the location of the operands. In general, this assignment is non-trivial.

3.2 Prefix Operations

The *prefix operation* takes a binary associative operator $\oplus$ and an array of n elements $x_1,\ldots,x_n$ and computes the array $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n$. For instance, if $\oplus$ is addition, then the prefix operation applied to the numbers 7,4,3,0,-1,4,2 would return 7,11,14,14,13,17,19. The prefix operation with addition is called *prefix-sums* operation. As for another example, let $\oplus$ be the binary minimum operation. Then, the prefix-minima operation applied to 7,4,3,0,-1,4,2 returns 7,4,3,0,-1,-1,-1.

In what follows, we describe a recursive parallel function that implements the prefix operation [LF80]. Let $x_1,\ldots,x_n$ be the input. Without loss of generality, we assume that n is a power of 2, *i.e.*, $n = 2^k$, for some nonnegative integer k. If $n = 1$ we return the input. Otherwise, we compute the array $x_1 \oplus x_2, x_3 \oplus x_4, \ldots, x_{n-1} \oplus x_n$ and apply recursively our prefix operation to this array. Let $y_1,\ldots,y_{n/2}$ denote the result. Finally, we return the array $z_1,\ldots,z_n$, where z_i is computed as follows: For $i = 1$, we set $z_1 := x_1$. For even i, we set $z_i := y_{i/2}$, and for odd $i > 1$, we set $z_i := y_{(i-1)/2} \oplus x_i$. It is not difficult to see that $z_i = \oplus_{j=1}^{i} x_j$.

We assume that $\oplus$ can be evaluated by one processor in constant time. Since in every recursive call to our function the size of the argument is halved, the number of recursive calls is $\log n$. Furthermore, each incarnation of our function can be performed without concurrent memory access in constant parallel time on n processors (not counting the recursive calls). Hence, the algorithm runs in time $O(\log n)$ on n processors. To reduce the number of processors, we apply Brent's scheduling principle. Instead of n processors we use only $n/\log n$ processors. Consider the work $W_j(n)$ performed by the algorithm in incarnation j, not counting the recursive call. For some suitable positive constant c, the work performed in the first incarnation is cn, in the second incarnation this work is reduced to $cn/2$, and so on. In general, in incarnation j $(j = 1,\ldots,\log n)$, $W_j(n) = c2^{k-j+1}$. It follows that the total work performed by the algorithm is $W(n) = \sum_{j=1}^{\log n} W_j(n) = O(n)$. On $n/\log n$ processors we can simulate incarnation j in time $\lceil W_j(n)\log n/n\rceil$. Since there are $\log n$ incarnations, the total running time is

$$\sum_{i=1}^{\log n}\left\lceil\frac{W_j(n)\log n}{n}\right\rceil \le \sum_{i=1}^{\log n}\left(\frac{W_j(n)\log n}{n}+1\right) = O(\log n).$$

Due to the regular structure of the prefix algorithm Brent's scheduling principle can be applied. It is not difficult to allocate work to processors in each incarnation, but since the details do not provide new insight, we omit them. We have shown the following:

Theorem 3.1 ([LF80]) *A prefix operation on n elements can be performed on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

Prefix operations are quite useful. Some basic applications of prefix operations are given in the following.

On the EREW PRAM, prefix operations are often used to distribute data from one memory location to others, *e.g.*, to simulate concurrent read accesses to memory. Suppose we want to broadcast a value x from cell 1 to the cells $2,\ldots,n$. First, we write a zero into cells $2,\ldots,n$. Then we execute a prefix-sums operation on the array from cell 1 to n. After that, each of the n memory locations has value x. Clearly, on PRAMs capable of concurrent read, we do not require prefix operations to distribute data.

Another application is the extraction of subsequences. Suppose we are given an array $X = x_1,\ldots,x_n$ of arbitrary elements and an array $A = a_1,\ldots,a_n$ with $a_i \in \{0,1\}$. To extract the subsequence Y of X that consists of all elements x_i with $a_i = 1$, we compute the prefix-sums of A and store x_i into a new array Y at position $\sum_{j=1}^{i} a_j$ if $a_i = 1$.

3.2.1 Segmented Prefix Operations

Suppose we are given an array $X = x_1,\ldots,x_n$ of elements drawn from a set W and a Boolean array $B = b_1,\ldots,b_n$ that splits X into a number of blocks in the following sense. From left to right, each entry of B that is true marks the beginning of a new block of X. Hence, if b_i is the k-th *true* entry of B and b_j is the $(k+1)$-th *true* entry of B, then the k-th block of X is $a_i,\ldots,a_{j-1}$. The last block of X starts with the index of the last *true* entry of B and ends with x_n. Our goal is to apply a prefix operation independently and simultaneously to each block of X (as defined by B). As it turns out, this *segmented prefix operation problem* can be solved by a single prefix operation on X, provided that we carefully choose the binary prefix operator.

Let $\oplus$ be the binary associative operator that we want to apply to the elements of X. We define a new binary operator $\otimes$ on $W \times \{\mathit{true}, \mathit{false}\}$ as follows:

$$(x,b) \otimes (y,c) := \begin{cases} (x \oplus y, b) & \text{if } c = \mathit{false}, \\ (y,c) & \text{otherwise.} \end{cases}$$

It is not difficult to see that $\otimes$ is associative, provided that $\oplus$ is associative. As before, let the k-th block of X span the indices i to $j-1$, and let q be some index between i and $j-1$. Then the first component of $\otimes_{p=1}^{q}(x_p, b_p)$ equals $\oplus_{p=i}^{q} x_p$. Hence, a single prefix-$\otimes$ operation on the tuples $(x_1,b_1),\ldots,(x_n,b_n)$ computes a prefix-$\oplus$ operation in each block of X.

Theorem 3.2 *A segmented prefix operation on an input of size n can be performed on the EREW PRAM in $O(\log n)$ time using $n/\log n$ processors.*

3.3 Computing the Maximum in Constant Time

Consider the problem of computing the maximum of n numbers. If we apply a prefix-maxima operation to the array of numbers, then the prefix maximum for the last number is the maximum of all numbers. Hence, we require $O(\log n)$ time and use $n/\log n$ processors. Clearly, this is a work optimal algorithm, since we have to perform at least $O(n)$ operations to look at all numbers. On the CREW PRAM (and EREW PRAM) this algorithm is also time optimal [CDR86], but on the CRCW PRAM we can do better. In the following we assume the weakest variant of the CRCW PRAM, namely, the common CRCW PRAM, where all processors that write to the same memory cell at the same time have to write the same value.

It has been shown by Shiloach and Vishkin [SV81] that for any $\varepsilon > 0$, the maximum (or minimum) of n elements can be found on the CRCW PRAM in constant time using $n^{1+\varepsilon}$ processors. We will show in the following that if all numbers are integers in the range $0,\ldots,O(n)$, the number of processors can be reduced to n, while still running in constant time. The algorithm exploits the fact that the OR of n Boolean values can be computed on the CRCW PRAM with n processors in constant time.

We first consider a problem related to the maximum problem. Given a Boolean array $X = x_1, \ldots, x_n$ that contains at least one *true*, determine the largest index i such that $x_i = \mathit{true}$. Assume that $\binom{n}{2} = O(n^2)$ processors are available. We create a Boolean array $Y = y_1, \ldots, y_n$, with all entries initially set to *false*. Each of the $\binom{n}{2}$ processors selects a distinct pair of indices i and j, with $1 \leq i < j \leq n$, and writes *true* into y_i iff $x_j = \mathit{true}$ or $x_i = \mathit{false}$. After that, exactly one entry y_k contains *false*, and this k is the largest index with $x_k = \mathit{true}$. All of these operations can be performed in constant time.

Now, we assume that only n processors are available. We split X into approximately $\sqrt{n}$ blocks $X_1, \ldots, X_{\sqrt{n}}$, each of size roughly $\sqrt{n}$. For each block X_i, we compute the Boolean OR of the bits in X_i and then use the algorithm of the previous paragraph to determine the last block X_k that contains a *true*. Since there are $\sqrt{n}$ blocks, this can be done on n processors in constant time. Finally, we determine the largest index j in block X_k with $x_j = \mathit{true}$, again using the algorithm given in the previous paragraph. Only $\sqrt{n}$ elements have to be handled, hence constant time and n processors suffice.

Let us return to our original problem. Let $a_1, \ldots, a_n$ denote the input with numbers in the range $0, \ldots, f(n)$, with $f(n) = O(n)$. We create a Boolean array $b_0, \ldots, b_{f(n)}$ and set b_j to *true* iff there exists an index i with $a_i = j$. This array can be created in constant time on n processors, since $f(n) = O(n)$. All we have to do now is to find the largest index i with $b_i = \mathit{true}$. As we have seen above, this can be done in constant time on $f(n)$ processors.

Theorem 3.3 ([FRW88]) *The maximum of n integers in the range* $0, \ldots, O(n)$ *can be computed on the (common) CRCW PRAM in constant time using n processors.*

3.4 Merging

Given two sorted sequences X and Y, the problem of merging X and Y is to compute a new sorted sequence that consists of the elements of X and Y. We first describe a non-optimal algorithm for merging based on bitonic sequences. Then a work optimal algorithm is derived that uses the bitonic merger as a subroutine.

3.4.1 Bitonic Merge

A sequence of elements drawn from a linearly ordered set is called *bitonic* if it is a concatenation of two monotonic sequences, one nondecreasing and the other nonincreasing. We say that the sequence remains bitonic if it is split somewhere and the two parts are interchanged. For example, the sequence $7,4,9,20,33,10$ is bitonic since $9,20,33$ is nondecreasing and $10,7,4$ is nonincreasing. An interesting property of bitonic sequences is the following: If $x_1,\ldots,x_{2n}$ is bitonic, then the sequences

$$\min(x_1,x_{n+1}),\min(x_2,x_{n+2}),\ldots,\min(x_n,x_{2n}) \tag{3.1}$$

and

$$\max(x_1,x_{n+1}),\max(x_2,x_{n+2}),\ldots,\max(x_n,x_{2n}) \tag{3.2}$$

are bitonic as well, and no element of (3.1) is greater than any element of (3.2).

To see this, consider the following. If $x_1,\ldots,x_{2n}$ is split and the two parts are interchanged, then the sequences (3.1) and (3.2) are split and interchanged the same way. Hence, it suffices to look at the case where

$$x_1 \leq \cdots \leq x_{j-1} \leq x_j \geq x_{j+1} \geq \cdots \geq x_{2n},$$

for some j, $1 \leq j \leq 2n$. We can furthermore assume that $j > n$, since the reversal of sequences does not change their bitonic property. If $x_n \leq x_{2n}$ then $x_i \leq x_{n+i}$, for all $1 \leq i \leq n$. Hence, sequence (3.1) equals $x_1,\ldots,x_n$ and sequence (3.2) equals $x_{n+1},\ldots,x_{2n}$, which are both bitonic sequences. Otherwise $x_n > x_{2n}$. Then there exists an index i, $1 \leq i \leq n$ such that $x_i > x_{n+i}$ and $x_{i-1} \leq x_{n+i-1}$. It follows that the sequence (3.1) equals $x_1,\ldots,x_{i-1},x_{n+i},\ldots,x_{2n}$ and the sequence (3.2) equals $x_{n+1},\ldots,x_{n+i-1},x_i,\ldots,x_n$ (cf. Figure 3.1). It is not difficult to see that no element of (3.1) is greater than any element of (3.2) and that both sequences are bitonic.

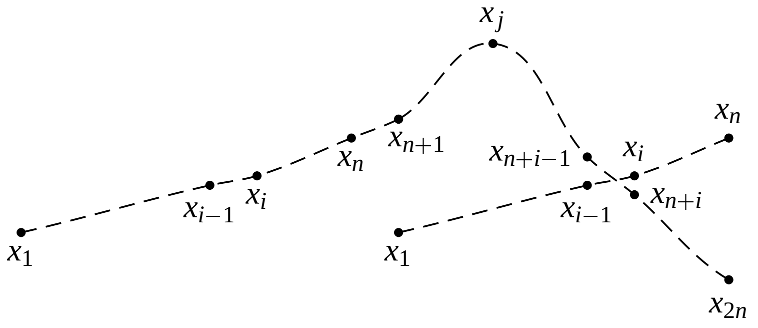


Figure 3.1: *Bitonic sequences (see text).*

Using this property of bitonic sequences, we can design a simple recursive function that sorts a bitonic sequence [Bat68]: Without loss of generality, we assume that $n = 2^k$, for some positive integer k. Let $x_1,\ldots,x_n$ be the input sequence. If $n = 1$ we return the input. Otherwise, we recursively sort the sequence $\min(x_1,x_{n/2+1})$, …, $\min(x_{n/2},x_n)$ and the sequence $\max(x_1,x_{n/2+1})$, …, $\max(x_{n/2},x_n)$. In the end, we return the concatenated results of the two recursive sorts.

With input size n, an incarnation of this function (not counting the recursive calls) can be performed in a constant number of parallel steps on an n processor EREW PRAM. The two recursive calls are processed in parallel: One half of the processors sorts the sequence of minima and the other half sorts the other sequence. Hence, the running time $T(n)$ of our function satisfies the following recurrence: $T(n) = T(n/2) + c$, for some positive constant c. The solution of this recurrence is $T(n) = O(\log n)$.

Now, let x_1, …, x_n and y_1, …, y_m be two sorted sequences. Then the sequence x_1, …, x_n, y_m, …, y_1 is bitonic, and we can use our function that sorts a bitonic sequence to merge the sequences x_1, …, x_n and y_1, …, y_m. We obtain:

Theorem 3.4 ([Bat68]) *Two sorted sequences of total length n can be merged on an n processor EREW PRAM in time $O(\log n)$.*

3.4.2 A Work Optimal Merging Algorithm

The bitonic merger just described is not work optimal, since one processor can merge two sorted sequences in $O(n)$ time but the bitonic merger requires $O(n \log n)$ operations. In the following we describe a work optimal merging algorithm. Our description mainly follows [AMW89]. Similar algorithms can be found in [BN89, HR89].

Let $X = x_1,\ldots,x_r$ and $Y = y_1,\ldots,y_s$ be two sorted sequences, and let $n = r + s$. We assume that all elements in X and Y are distinct. If this is not the case, distinctness can be achieved as follows. We replace each element x_i in the first sequence by a triple $(x_i,1,i)$ and replace each element y_i in the second sequence by a triple $(y_i,2,i)$. Then, we merge the two sequences of triples using lexicographic order.

Let v be an element. We say the *rank* of v in X is k if exactly k elements of X are less than v. Let $X = X_1RX_2$ and $Y = Y_1SY_2$, *i.e.*, R is a (possibly empty) subsequence of X and S is a (possibly empty) subsequence of Y and X_1,X_2,Y_1,Y_2 are the respective prefixes and suffixes of X and Y. We say R and S are *compatible* if all elements of R and S are greater than all elements of X_1 and Y_1 and less than all elements of X_2 and Y_2. For instance, let $X = 2, 5, 7, 8, 10, 14, 22, 30$ and $Y = 1, 3, 6, 12, 15, 34, 35$. Then the subsequences $8, 10, 14$ and $12, 15$ are compatible. Furthermore, $7, 8, 10$ and $12, 15$ are compatible, and $7, 8$ is compatible with the empty sequence in Y between 6 and 12. Given a subsequence R of X, a subsequence of Y compatible with R can be determined as follows. Take the first and the last element of R and compute their rank in Y. Let a be the rank of the first element and let b be the rank of the last element. Then the subsequence of Y consisting of the $(a+1)$-th up to the b-th element of Y is compatible with R.

Assume for the time being that we have partitioned X and Y into $O(n/\log n)$ pairs of compatible subsequences R_j and S_j such that for all j, R_j is a subsequence of X, S_j is a subsequence of Y, and R_j, S_j consist of $O(\log n)$ elements each. Obviously, we can merge X and Y by first merging each pair R_j, S_j and then concatenating the results for all j. Using one processor for each pair R_j and S_j, we can perform all merges in time $O(\log n)$ using $n/\log n$ processors.

What remains is to compute the R_j's and S_j's. From X and Y, we select every $\lceil \log n \rceil$-th element and merge the two selected subsequences using bitonic merge. By Theorem 3.4, this can be performed in time $O(\log n)$ using $n/\log n$ EREW PRAM processors. Let $u^{(1)}$, $u^{(2)}$, … be the selected elements of X, and let $v^{(1)}$, $v^{(2)}$, … be the selected elements of Y. Furthermore, let $U^{(i)}$ be the subsequence of X starting with $u^{(i)}$ and ending right before $u^{(i+1)}$, *i.e.*, $U^{(i)}$ contains $u^{(i)}$ but not $u^{(i+1)}$. Similarly, let $V^{(i)}$ be the subsequence of Y starting with $v^{(i)}$ and ending right before $v^{(i+1)}$.

If we know the rank of each $v^{(j)}$ in X and the rank of each $u^{(i)}$ in Y, then it is easy to compute pairs of compatible R_j's and S_j's with the desired properties (Figure 3.2). In the following we describe how to compute the rank of each $v^{(j)}$ in X. The same algorithm can be used to compute the ranks of the $u^{(i)}$'s in Y.

Let W be the merged sequence of $v^{(j)}$'s and $u^{(i)}$'s. Using a prefix operation on W segmented by the $u^{(i)}$'s, we can determine for each $v^{(j)}$ the index i such that $v^{(j)}$ falls into $U^{(i)}$. Then, using binary search, we determine the exact position in $U^{(i)}$ where $v^{(j)}$ belongs. One processor can perform the binary search in time $O(\log\log n)$. If more than one element $v^{(j)}$ falls within the same $U^{(i)}$, then concurrent read accesses to the same memory cell may occur. To avoid this, we make a sufficient number of copies of $U^{(i)}$ beforehand, and let each binary search use a distinct copy. Let $v^{(j)}, \ldots, v^{(k)}$ be the elements that fall into the same subsequence $U^{(i)}$. We require $k - j + 1$ copies of $U^{(i)}$ and we create these copies performing $|U^{(i)}|$ prefix-maxima operations each on a

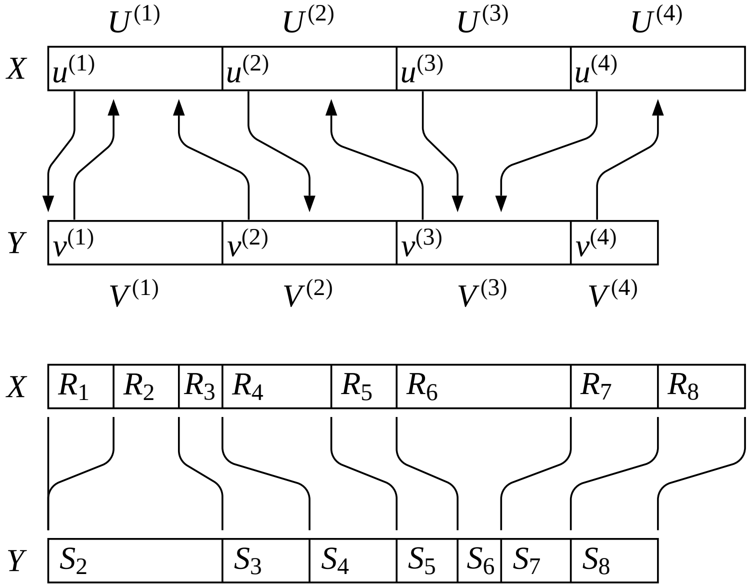


Figure 3.2: *To merge X and Y we determine the rank of each $u^{(i)}$ in Y and the rank of each $v^{(j)}$ in X. We obtain pairs of "small" compatible subsequences R_j and S_j.*

sequence of size $k-j+1$. We use $k-j+1$ processors and require $O(\log n)$ time because the size of $U^{(i)}$ is $O(\log n)$. Since the number of selected elements $v^{(j)}$ is $O(n/\log n)$, the number of binary searches that have to be performed is $O(n/\log n)$ too. Hence, we can determine the ranks of all $v^{(j)}$ in X in time $O(\log n)$ using $n/\log n$ EREW PRAM processors.

We have just shown:

Theorem 3.5 ([AMW89, BN89, HR89]) *Two sorted sequences of total size n can be merged on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

3.4.3 Segmented Merging

Assume we are given k pairs of sorted sequences X_i and Y_i, $i=1,\ldots,k$, and we are supposed to merge X_i with Y_i, for every i. Instead of performing k merge operations, we perform only one. In each X_i we replace every element x by a tuple (i,x). Then we concatenate $X_1,\ldots,X_k$ to form the sequence A. We carry out the same operations on the Y_i's to form the sequence B. We merge A

and B using lexicographic order on the tuples, and thus obtain a sequence that consists of X_1 merged with Y_1, followed by X_2 merged with Y_2, and so on.

Theorem 3.6 *We can merge any number of pairs of sorted sequences with total length n in $O(\log n)$ time using $n/\log n$ EREW PRAM processors.*

3.5 Sorting

Let X be a sequence of n elements drawn from a linearly ordered set. Without loss of generality, we assume $n = 2^k$, for some positive integer k. To sort X we can use *merge-sort*, which works as follows. First, we split X into n sequences of length 1. Now, we iterate $\log n$ times. In each iteration we group the set of sorted sequences obtained so far into pairs and merge each pair into a new sorted sequence. In doing so, the number of sequences is halved, while their length doubles. Consequently, $\log n$ iterations suffice to obtain a single sorted sequence Y that consists of all elements of X. All merge operations of one iteration are carried out simultaneously using segmented merging. Hence, we can prove:

Theorem 3.7 *A sequence of n elements drawn from a linearly ordered set can be sorted on the EREW PRAM in time $O(\log^2 n)$ using $n/\log n$ processors.*

If we are only interested in a sorted sequence that consists of the s smallest elements of X, then we can perform better. We assume that n and s are both powers of 2. We divide merge-sort into two phases. Phase one consists of $\log s$ iterations resulting in n/s sorted sequences of size s each. We require $O(\log n \log s)$ time on $n/\log n$ processors to perform phase one. In phase two, every time we merge two sequences of size s we only keep the first s elements of the result. This way the number of elements we have to handle is halved in each iteration. Since all sequences are of size s, each merge requires $O(s)$ operations and takes $O(\log s)$ time, provided that we use at least $s/\log s$ processors. In iteration i, we have to merge $n/(s2^{i-1})$ sequences and the work performed is $O(n/2^{i-1})$. If we use $n/\log n$ processors we can perform iteration i in time $O(\max(\log s, (\log n)/2^{i-1}))$. The total time spent in phase two is $O(\log n \log s)$ since at most $\log n$ iterations have to be performed. We obtain:

Theorem 3.8 *Let X be a sequence of n elements drawn from a linearly ordered set, and let s be an integer with $1 \leq s \leq n$. A sorted sequence that consists of the s smallest elements of X can be computed in time $O(\log n \log s)$ using $n/\log n$ EREW PRAM processors.*

The merge-sort algorithm given at the beginning of this section is work optimal but not time optimal. A work optimal sorting algorithm for the EREW PRAM that is also time optimal is Cole's pipelined merge-sort:

Theorem 3.9 ([Col88]) *A sequence of n elements can be sorted in $O(\log n)$ time using n EREW PRAM processors.*

3.6 List Ranking

Suppose we are given a linked list L of n nodes whose order is specified by an array of pointers $s(1),\ldots,s(n)$: in L, the node following node i has number $s(i)$, for $1 \leq i \leq n$. We assume that the first node of L is node 1 and if node i is the last node, then $s(i) = 0$. The list ranking problem is to determine the rank of each node, *i.e.,* its distance to the end of the list.

On a sequential computer, list ranking is a trivial problem. All we have to do is to traverse the list. In the parallel setting, list ranking is more involved. A simple list ranking algorithm is the following, which uses the *pointer jumping* technique. To each node we dedicate one processor and a variable $d(i)$ that will contain the distance of node i to the end of the list. Initially, $d(i)$ is zero if node i is the last node of the list and otherwise $d(i) = 1$. Then, we iterate the following: In each iteration, processor i executes $d(i) := d(i) + d(s(i))$ and $s(i) := s(s(i))$ if $s(i)$ is not yet zero. After $O(\log n)$ iterations all pointers are zero and $d(i)$ is the distance of node i to the end of the list (Figure 3.3). Each iteration requires a constant number of parallel steps on the EREW PRAM since no concurrent memory accesses occur. Hence, we obtain:

Theorem 3.10 ([Wyl79]) *Given a linked list L with n nodes, the rank of each node in L can be determined on the EREW PRAM in $O(\log n)$ time using n processors.*

Clearly, this algorithm is not work optimal. Work optimal list ranking algorithms for the EREW PRAM are known but are quite involved.

Theorem 3.11 ([AM88, CV88a]) *The list ranking problem for a list of size n can be solved in $O(\log n)$ time on $n/\log n$ EREW PRAM processors.*

The basic idea behind these algorithms is the following strategy:

1. Shrink the list until only $O(n/\log n)$ nodes remain.

2. Apply pointer jumping to the list of remaining nodes.

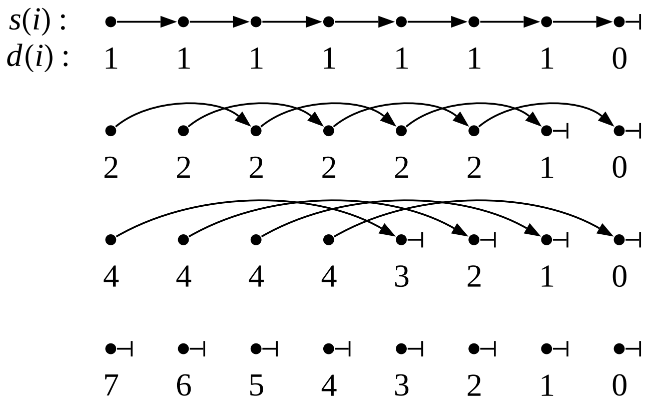


Figure 3.3: *List ranking using pointer jumping. After* $\lceil \log n \rceil$ *iterations (here: 3), all nodes know their distance to the end of the list.*

3. Restore the original list and rank the nodes that have been removed.

The difficult part is step 1. Step 2 has just been described and step 3 is essentially step 1 reversed. For details, we refer to [AM88] and [CV88a].

3.6.1 Prefix Operations on Linked Lists

List ranking can be used to perform a prefix operation on the elements in a linked list. To this end, we determine the rank $d(i)$ of each list node i and store the element of node i in an array X at position $n-d(i)$, where n is the length of the given list. Next, we perform the desired prefix operation on X. Finally, we move the prefix value computed in X at position $n-d(i)$ back to node i. Using Theorem 3.11 and Theorem 3.1 we obtain:

Theorem 3.12 *A prefix operation on the elements of a linked list can be computed in time* $O(\log n)$ *using* $n/\log n$ *EREW PRAM processors.*

3.7 The Euler-Tour Technique

A number of important problems on trees can be solved using the Euler-tour technique, which has been introduced by Tarjan and Vishkin [TV85]. In the following we restrict our attention to rooted trees. We assume that the given tree is ordered, that is, for each vertex an explicit order of its children is given.

For example, the tree may be represented by a sequence of vertices, each vertex being associated with a pointer to its right sibling and a pointer to its parent. This is a necessary prerequisite for the Euler-tour technique. If the tree is not ordered, we have to order the children of each vertex using sorting.

Every edge (x,y) of the given tree is replaced by two anti-parallel arcs $d(x,y)$ and $u(x,y)$, one pointing downwards from x to y and the other pointing upwards from y to x. We build a list of these arcs such that the resulting path runs counterclockwise along the contour of the tree, *i.e.*, the path is an Eulerian tour of the tree. If the number of vertices in the tree is n, this can be performed in constant time on n processors since the tree is ordered. We break this 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. We call this path the *Euler-tour* of the given tree. In Figure 3.4, a tree and its Euler-tour are depicted.

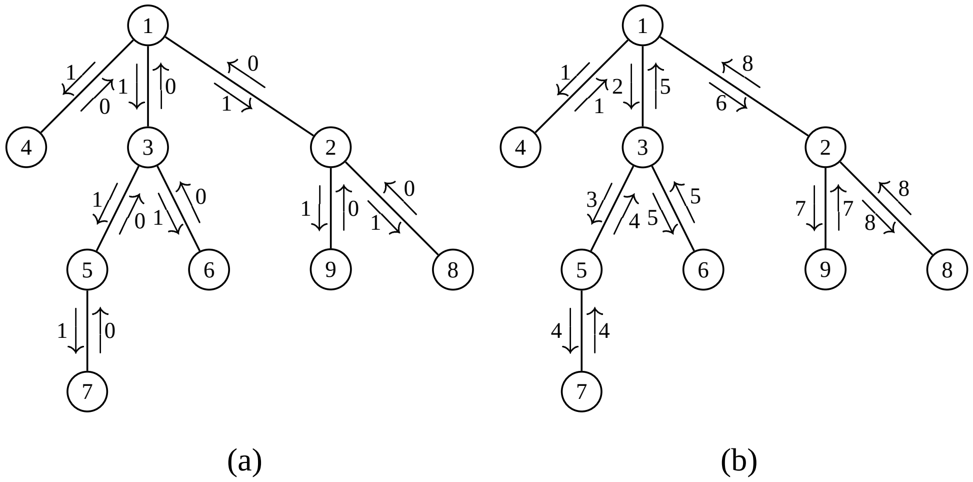


Figure 3.4: *A tree and its Euler-tour* $d(1,4)$, $u(1,4)$, $d(1,3)$, $d(3,5)$, $d(5,7)$, $u(5,7)$, $u(3,5)$, $d(3,6)$, $u(3,6)$, …, $u(2,8)$, $u(1,2)$. *(a) To compute the preorder numbers of the tree vertices, we assign 1 to downgoing arcs of the Euler-tour and 0 to upgoing arcs. (b) The prefix sums of the arc values on the Euler-tour. The prefix sum for arc* $d(x,y)$ *is the preorder number of* y *minus 1.*

In the following we show how the Euler-tour can be used to compute the preorder number and the depth of each vertex in the given tree. We start with the depth. We assign a value of -1 with each upgoing arc in the Euler-tour and a value of 1 with each downgoing arc. Then, we compute the prefix sums of the arc values on the Euler-tour. The prefix sum obtained this way for $d(x,y)$ is the depth of y minus 1. To compute preorder numbers, we proceed the same way as just described, with the exception that a value of 0 is assigned to each upgoing arc of the Euler-tour (instead of -1). Now the prefix sum obtained for $d(x,y)$ is the preorder number of y minus 1 (Figure 3.4b).

Theorem 3.13 *Given an ordered tree with n vertices, we can compute the depth and the preorder number of each vertex in time $O(\log n)$ using $n / \log n$ processors of an EREW PRAM.*

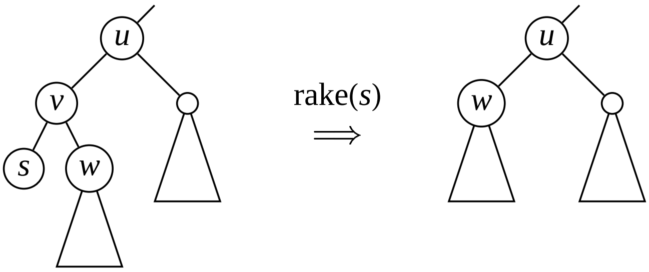
Proof. The Euler-tour can be constructed in constant time on n processors, and hence in $O(\log n)$ time on $n / \log n$ processors. A prefix-sums operation on the elements of a linked list can be carried out within the same bounds (Theorem 3.12). □

3.8 Tree Contraction

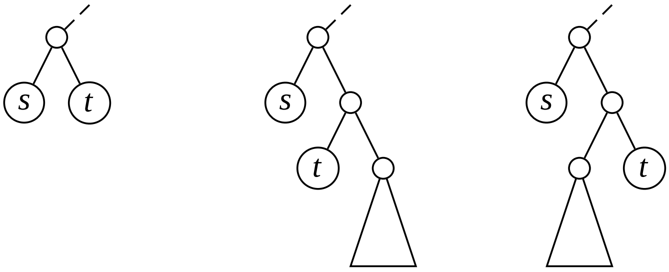
Suppose we are given a binary tree where to each leaf some value is assigned, each internal vertex of the tree represents a binary function, and each edge represents a unary function. The value of an edge is determined by applying its function to the value of the child it points to. The value of a vertex is determined by applying its function to the values of the edges that point to its two children. We call such a binary tree an *expression tree*. The goal is to compute the values of all vertices in the given expression tree. An efficient way of solving this expression evaluation problem in parallel is *tree contraction*. This technique has been introduced by Miller and Reif in [MR85]. Since then a number of authors contributed to it [KD88, CV88b, ADKP89, GR89]. Our description follows [KD88] and [ADKP89].

The basic idea of tree contraction is to contract the given binary tree by iteratively removing vertices until only three vertices remain: its root and two of its leaves. During contraction, the values of the removed leaves and the functions of the removed vertices and edges are combined into new functions, and these functions are associated with the remaining edges. After each contraction step, the value of each of the remaining vertices, as defined by the new expression tree, is identical to its value defined by the original expression tree. Hence, at the end of contraction the value of the root can easily be computed. To compute the values of the removed vertices, the final expression tree is expanded to its original size by undoing the sequence of contraction steps in reverse order.

In the following we first show how vertices and edges are removed from the tree during contraction. We will later show how to handle the functions they represent. The basic operation used to contract the tree is *raking*. A rake operation takes a leaf s, its parent v, its sibling w, and the parent u of its parent. It removes s and v from the tree and the edges incident to them and puts a new edge from u to w:



In order to be fast, we want to rake in each parallel step a large number of leaves simultaneously. In the following three situations the two leaves s and t can not be raked simultaneously:



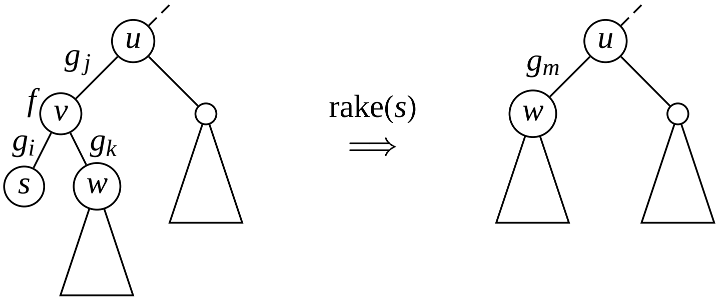
These (and their symmetric counterparts) are the only situations where two rake operations interfere with each other.

Our strategy for scheduling the rake operations is the following. First, we number the leaves counterclockwise along the contour of the binary tree. This can easily be done using the Euler-tour technique. Then we iterate until only two leaves remain. In each iteration, we first simultaneously rake all odd numbered left leaves and then we simultaneously rake all odd numbered right leaves. (The only exception is the first and the last leaf: if one of them is a child of the root, we can not rake it. Instead, we do not touch this leaf anymore, since it will be one of the two leaves that remain till the end.) Finally, we divide the number of each of the remaining (even numbered) leaves by 2, and start the next iteration.

It is not difficult to see that the simultaneous rakes do not interfere with each other. Moreover, in each iteration the number of leaves that remain to be raked is at least halved. Hence, after $O(\log n)$ iterations the tree is reduced to a tree that consists of the root and two leaves. Each iteration can be performed on n processors in a constant number of parallel steps. In the first iteration $O(n)$ work is performed, and in each further iteration, only half of the work of the previous iteration needs to be done. Hence, the total number of operations required to contract the tree is $O(n)$. To reduce the number of processors from n to $n/\log n$, we apply Brent's scheduling principle: On $n/\log n$ processors the time required to contract the tree is still $O(\log n)$.

Lemma 3.14 *A binary tree of size n can be contracted on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

We now return to the problem of computing the values of the vertices in the given expression tree. We first focus on how the value of the root of the tree can be computed. We assume that the functions associated with edges are taken from an indexed set of unary functions. In particular, each function is uniquely determined by an index. Suppose we want to rake a leaf s with value val(s). As before, let v be the parent of s, let u be the parent of v, and let w be the sibling of s. Furthermore, let f be the function associated with v and let g_i, g_j, and g_k be the functions associated with the edges (v,s), (u,v), and (v,w). We rake s as described before and additionally, we assign a new function g_m to the new edge (u,w):



The new function g_m is defined as

$$g_m(x) = g_j(f(g_i(\text{val}(s)), g_k(x))). \tag{3.3}$$

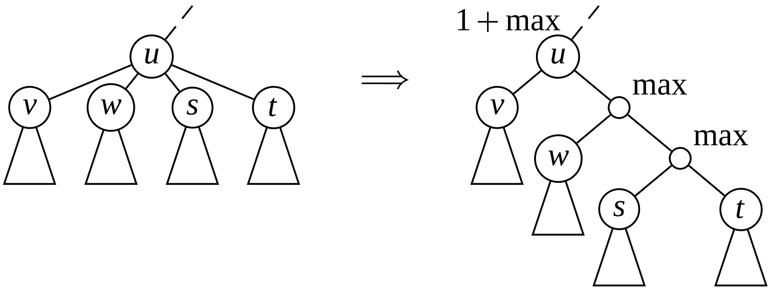
Hence, by raking s, the values of all remaining vertices remain unchanged. In particular, the value of the root in the contracted expression tree is still its value as defined in the original expression tree. We obtain the following result: If the index m can be determined from f, i, j, and k in constant sequential time and if all edge functions can be evaluated in constant sequential time, given their index and argument, then the value of the root can be computed in $O(\log n)$ time using $n/\log n$ processors.

What remains is to compute the values of all other non-leaf vertices. To this end, we undo all rake operations in reverse order. We start with the contracted tree that consists of the root and two leaves and end up with the original expression tree. Assume that leaf s has to be unraked in the next iteration of unraking. Furthermore, assume that all values of vertices in the current expression tree are computed. To unrake s, we remove the edge (u,w) and reinsert v and s together with their incident edges (u,v), (v,s), and (v,w). Since the value of w is known by now, we can compute the value of v, which is $f(g_i(\text{val}(s)), g_k(\text{val}(w)))$. Clearly, "uncontracting" the tree and computing all values can be performed within the same time and processor bounds as contracting the tree.

Tree contraction can be applied to a wide range of expression evaluation problems. In each individual case we have to provide a suitable set of indexed edge functions. In many cases, tree contraction can also be used to solve expression evaluation problems on arbitrary (non-binary) trees. The idea is to convert the arbitrary input tree into a binary tree with an expression evaluation problem that is equivalent to the original problem.

3.8.1 Computing the Vertex Height in Trees

In the following we show how tree contraction can be used to determine the height of each vertex in an arbitrary ordered tree. To each leaf we assign a value of 1 and to each internal vertex u we add auxiliary vertices as follows:



Each "old" vertex is associated with the function $f : (x,y) \mapsto 1 + \max(x,y)$ that adds 1 to the maximum of the values of its left and right child. Each auxiliary vertex is associated with the function $f' : (x,y) \mapsto \max(x,y)$. All edges obtain the identity function. It is not difficult to see that the value of an "old" vertex in the resulting binary expression tree equals its height in the original tree. Since the input tree is ordered, all auxiliary vertices can be inserted into the tree in constant time on n processors, and hence in $O(\log n)$ time on $n/\log n$ processors.

Suitable edge functions for tree contraction are the following. For integers a and b, let $g_{a,b}$ be the unary function

$$g_{a,b} : x \mapsto a + \max(b,x).$$

Initially, we let all edges carry the function $g_{0,0}$, since for positive x, $g_{0,0}$ is the identity function. Suppose that during tree contraction we have to rake a leaf s with parent v. As before, let u be the parent of v and let w be the sibling of s. Furthermore, let $g_{a,b}$ be the function of edge (v,s), let $g_{c,d}$ be the function of edge (u,v), and let $g_{e,h}$ be the function of (v,w). If the function of v is $1+\max$, then, according to (3.3), the edge function for the new edge (u,w) is supposed to be

$$\begin{aligned} & g_{c,d}(1 + \max(\underbrace{g_{a,b}(\mathrm{val}(s))}_{\overset{\mathrm{def}}{=} k}, g_{e,h}(x))) \\ = \quad & c + \max(d, 1 + \max(k, e + \max(h,x))) \\ = \quad & c + 1 + e + \max(\max(d-1-e, k-e, h), x) \\ = \quad & g_{c+1+e,\max(d-1-e,k-e,h)}(x), \end{aligned}$$

and if the function of v is max, then the edge function of (u,w) is

$$g_{c+e,\max(d-e,k-e,h)}(x).$$

In either case, the index of the new edge function can be computed in constant sequential time. It is furthermore clear that $g_{a,b}$ can be evaluated in constant sequential time, given its index (a,b) and its argument. We conclude that the functions $g_{a,b}$ are suitable functions for the edges during tree contraction. We have shown:

Theorem 3.15 *Using tree contraction, we can compute the height of each vertex in an ordered tree of size n in time $O(\log n)$ with $n/\log n$ EREW PRAM processors.*

3.8.2 Path Extraction

A problem that can be solved by tree contraction using the same edge functions as used in the previous subsection is *path extraction*. We are given an ordered tree with one designated leaf x and want to extract the path from x to the root. More precisely, the output is supposed to be an array that consists of the vertices of the path from x to the root, in the same order as the vertices appear on that path. To solve this problem, we use the same construction as in the previous subsection, with the following exceptions. To all leaves, except x, we assign value 0. To x we assign 1. We let all other vertices carry the function $f : (x,y) \mapsto \max(x,y)$. In the expression tree obtained this way, the value of a vertex is positive iff the vertex belongs to the path from x to the root of the tree. Now we can determine the rank of each vertex on that path (using list ranking) and store the vertices into an array.

Theorem 3.16 *Given an ordered tree and one designated leaf x, we can extract the path from x to the root of the tree in $O(\log n)$ time with $n/\log n$ EREW PRAM processors.*

In the following we show how this technique can be extended in order to handle a number of trees simultaneously. Suppose we are given trees $T_1,\ldots,T_k$, and k designated leaves $x_1,\ldots,x_k$, where x_i is a leaf in T_i. To compute the k paths that go from each x_i to the root of T_i, we proceed as follows. We join the trees $T_1,\ldots,T_k$ into a new tree T by adding a new vertex r and adding edges from the roots of $T_1,\ldots,T_k$ to r. To leaf x_i we assign the value i, while to all other leaves we assign 0. Then we follow the construction described above. In the resulting expression tree, the value of a vertex (other than r) is i iff the vertex is on the path from x_i to the root of T_i. We concatenate these paths into a single list and store the resulting list into an array. These last operations can be carried out by prefix operations and list ranking. We have shown:

Theorem 3.17 *Let $T_1,\ldots,T_k$ be ordered trees with a total number of n vertices, and let x_i be a designated leaf in T_i, for $i = 1,\ldots,k$. We can extract for all x_i simultaneously, the path from x_i to the root of T_i in time $O(\log n)$ using $n/\log n$ EREW PRAM processors.*

3.9 Parenthesis Matching

A string s of opening and closing parentheses is called well formed if s can be derived by the grammar

$$S \to \varepsilon \,\big|\, (S) \,\big|\, SS$$

A *matching pair* of parentheses in s is a pair that is generated in a single step in the derivation of s. For instance, the string (()(()())) is a well formed word of parentheses, while the string (()(()) is not well formed. In a well formed word of parentheses, every parenthesis is one half of a distinct matching pair. In (()()), the second and the third parenthesis, the fourth and the fifth, and the first and the last are matching pairs. The *parenthesis matching problem* is to determine all matching pairs in a given well formed word of parentheses. Early work on parallel algorithms for parenthesis matching was done in the context of parsing arithmetic expressions [DS83, BOV85]. Work optimal algorithms for parenthesis matching on the EREW PRAM have been proposed in [AMW89, CD91, TLC89]. Our description mainly follows [AMW89].

The key idea for efficient parallel parenthesis matching is to pipeline operations along a balanced binary computation tree. We take $n/\log n$ processors and let each of them be responsible for computations at a distinct vertex in a balanced binary tree with $n/\log n$ vertices. The number of leaf processors is then approximately $n/(2\log n)$. We split the input string (of length n) into pieces of length $2\log n$ and let each of the leaf processors be responsible for one piece of the input. Later, a processor at vertex v in the computation tree will be responsible for a certain interval of the input, namely, the pieces assigned to leaves in the subtree rooted at v. Our computation proceeds in three phases.

In phase one, each of the leaf processors determines sequentially all matching pairs in the piece of the input it is responsible for. Using a stack, this can be done in time linear in the number of parentheses in each piece, *i.e.*, in $O(\log n)$ time. Then all parentheses matched so far are removed. What remains at each leaf, is a (possibly empty) sequence of unmatched closing parentheses followed by a (possibly empty) sequence of unmatched opening parentheses.

In phase two, we perform a bottom-up computation in the balanced binary tree. For each vertex v, we compute a triple (c,m,o), where c is the number of closing parentheses not matched in the subtree rooted at v, o is the number of opening parentheses not matched, and m is the number of matching pairs that have their opening parenthesis in the left subtree of v and their closing parenthesis in the right subtree of v. Given the triples (c_l,m_l,o_l) and (c_r,m_r,o_r) of the left and right child of v, we can compute the triple (c,m,o) of v as follows:

$$\begin{aligned} m &= \min(o_l, c_r), \\ c &= c_l + c_r - m, \\ o &= o_l + o_r - m. \end{aligned}$$

We start this computation at the leaves and proceed level by level up to the root. After $O(\log n)$ parallel steps, all triples are computed. Note that the triple of the root equals $(0,x,0)$, for some integer x, iff the input string is a well formed word of parentheses. Figure 3.5 illustrates the first two phases of parenthesis matching given an input string of length 64.

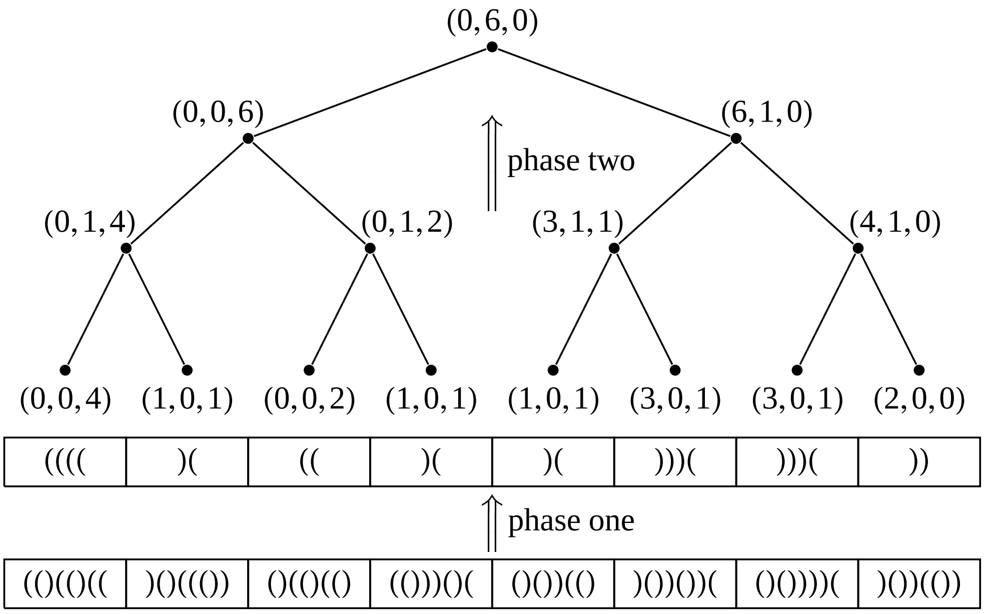


Figure 3.5: *The first two phases of parenthesis matching (see text).*

We now proceed to phase three. For each matching pair of parentheses, there exists a uniquely determined vertex v in the computation tree that is responsible for “matching” it, in the sense that the opening parenthesis of the pair belongs to the left subtree of v and the closing parenthesis of the pair belongs to the right subtree of v. If the triple of v is (c,m,o), then v is responsible for m such matching pairs. The idea is to let v assign a unique identifier to each of these m matching pairs. Then, v sends these identifiers to its left and right child. The children propagate these numbers to their children and so on, until the identifiers reach the leaves. If the propagation and distribution of identifiers is done carefully, then each identifier ends up at the right parenthesis. In particular, an identifier is delivered only to the closing and to the opening parenthesis of the matching pair that the identifier was assigned to at vertex v. In the end, the parentheses of a matching pair can find each other using the unique identifier.

In the sequel, we assume that the vertices in the computation tree are numbered and we refer to each vertex by its number. To obtain unique identifiers, we compute for each vertex v the value $b_v = \sum_{w<v} m_w$ using a prefix-sums operation on the m-values of vertices. Vertex v uses the numbers in the interval $[b_v, b_v + m_v - 1]$ to identify the m_v pairs matched at v. Thus, each number used to identify a matching pair is unique. Let us consider how the identifiers of one vertex v are propagated down to the leaves. The propagation of identifiers of the other vertices takes place at the same time in a pipelined fashion. Vertex v sends the message $(\mathrm{O}, [b_v, b_v + m_v - 1])$ to its left child and the message $(\mathrm{C}, [b_v, b_v + m_v - 1])$ to its right child, where O and C indicate the type of parentheses the identifiers belong to.

Let w be a descendent of v that receives the message $(\mathrm{O}, [a, b])$. If w is a leaf, then the identifiers in the interval $[a, b]$ are assigned sequentially to opening parentheses from right to left, *e.g.*, a is assigned to the rightmost opening parentheses at leaf w that has no identifier yet. Otherwise w has a left and a right child. Let (c, m, o) be the triple of w, and let (c_l, m_l, o_l) and (c_r, m_r, o_r) be the triples of the left and right child of w. The first $x = \min(o_r, b - a + 1)$ identifiers of the interval $[a, b]$ belong to opening parentheses in the right subtree of w and the remaining $y = b - a + 1 - x$ identifiers belong to opening parentheses in the left subtree of w. Hence, if $x > 0$ then w sends the message $(\mathrm{O}, [a, a + x - 1])$ to its right child and if $y > 0$ it sends the message $(\mathrm{O}, [a + x, b])$ to its left child. Finally, we update the triples of the left and right child of w:

$$o_r := o_r - x, \qquad o_l := o_l - y.$$

The other case, where w receives a message with identifiers for closing parentheses is handled symmetrically. If w is a leaf, then the numbers in $[a, b]$ are assigned sequentially to closing parentheses from left to right, *e.g.*, a is assigned to the leftmost closing parentheses at leaf w that has no identifier yet. If w is not a leaf, then the first $x = \min(c_l, b - a + 1)$ identifiers of the interval $[a, b]$ belong to closing parentheses in the left subtree of w and the remaining $y = b - a + 1 - x$ identifiers belong to closing parentheses in the right subtree of w. Therefore, if $x > 0$ then w sends the message $(\mathrm{C}, [a, a + x - 1])$ to its left child and if $y > 0$ it sends the message $(\mathrm{C}, [a+x, b])$ to its right child. Then, as before, we update the triples of the left and right child of w:

$$c_l := c_l - x, \qquad c_r := c_r - y.$$

The propagation of identifiers is done for all vertices v simultaneously in a pipelined fashion. In particular, all vertices v in the computation tree start to send out $(\mathrm{O}, [b_v, b_v + m_v - 1])$ and $(\mathrm{C}, [b_v, b_v + m_v - 1])$ at the same time. Since the height of the tree is $O(\log n)$ and each leaf contains $O(\log n)$ parentheses, all computations in the tree are finished after $O(\log n)$ parallel steps. In the end, each parenthesis (not matched in phase one) has a number that uniquely identifies the matching pair it belongs to. We have shown:

Theorem 3.18 ([AMW89, TLC89]) *The parenthesis matching problem can be solved on $n/\log n$ EREW PRAM processors in time $O(\log n)$.*

3.10 Breadth-First Traversal of Trees

The straightforward solution to the problem of computing the breadth-first traversal of an ordered tree is to compute the depth and the preorder number of each vertex, and to sort the vertices lexicographically by depth and preorder number. Sorting requires $O(n \log n)$ operations, and hence, we obtain an algorithm that is not work optimal because a sequential algorithm can perform an ordered breadth-first search in linear time. Chen and Das have shown that a breadth-first traversal can actually be computed by a work and time optimal EREW PRAM algorithm [CD92]. The algorithm combines the Euler-tour technique with parenthesis matching and works as follows.

We construct the Euler-tour of the given tree T and determine the depth of T. We assign to each downgoing arc in the Euler-tour a closing parenthesis. To each upgoing arc we assign an opening parenthesis. The string of parentheses that now runs along the Euler-tour is stored into an array, after we determined the position of each parenthesis using list ranking. This word of parentheses is not well formed, but by appending $\mathit{depth}(T) - 1$ closing parentheses and prepending $\mathit{depth}(T) - 1$ opening parentheses, we obtain a well formed word α of parentheses (Figure 3.6a). We compute the matching pairs in α and build a linked list of the parentheses of α as follows. We let an opening parenthesis point to its matching closing parenthesis. For $i = 1, \ldots, \mathit{depth}(T) - 2$, the i-th appended closing parenthesis points to the $(\mathit{depth}(T) - 1 - i)$-th prepended opening parenthesis. The last appended closing parenthesis will be the end of the list. Every other closing parenthesis belongs to a distinct edge of the tree and we let such a parenthesis point to the opening parenthesis that belongs to the same tree edge. As a result, we obtain a linked list of the parentheses of α that starts with the $(depth(T)-1)$-th prepended opening parenthesis and ends with the last appended closing parenthesis (Figure 3.6b).

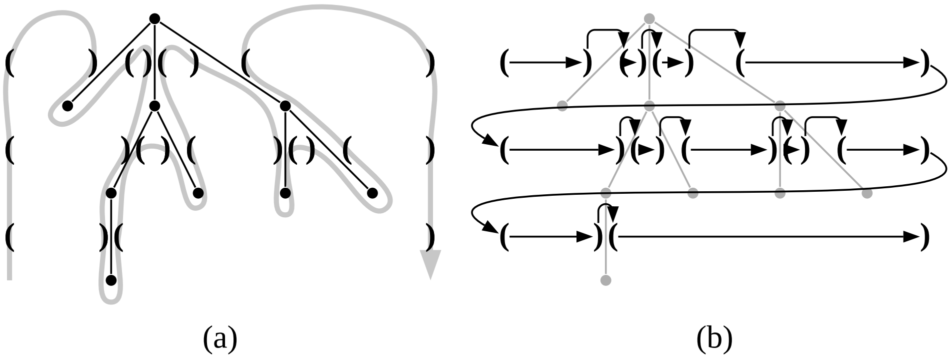


Figure 3.6: *Breadth-first traversal of a tree T. (a) The gray path indicates a well formed word α of parentheses, consisting of the parentheses assigned to arcs of the Euler-tour, $depth(T)-1$ prepended opening parentheses, and $depth(T)-1$ appended closing parentheses. (b) After computing the matching pairs in α, the parentheses are linked into a list that visits the edges in breadth-first order.*

From this list, we remove all but the closing parentheses that belong to edges. Finally, each of these closing parentheses is replaced by a vertex: The parenthesis that belongs to the downgoing arc $d(u,v)$ of the Euler-tour is replaced by v. We prepend the root of T and obtain a list of vertices that is the ordered breadth-first traversal of T.

Theorem 3.19 ([CD92]) *The breadth-first traversal of an ordered tree with n vertices can be computed on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

3.11 Algorithms for Vertex Series Parallel Digraphs

Vertex series parallel dags are the precedence graphs of series parallel orders. In the following we present elementary algorithms on VSP dags that are used in Chapter 6 to compute optimal two processor schedules for series parallel orders. We first show how a decomposition tree for a VSP dag can be computed using an algorithm that computes decomposition trees for ESP multidags. Then, we show how the heights of the defining subgraphs of a VSP dag with regard to a given decomposition tree can be determined. And finally, an algorithm is presented that computes the breadth-first traversal of a VSP dag.

3.11.1 Decomposition Trees

Parallel algorithms that recognize edge series parallel multidags and compute decomposition trees for them can be found in [HY87, Epp92, HH94, BdF96]. To compute a decomposition tree for a vertex series parallel dag G, we first have to compute its line digraph inverse $L^{-1}(G)$. $L^{-1}(G)$ is an ESP multidag and any decomposition tree for $L^{-1}(G)$ is also a decomposition tree for G (cf. Subsection 2.2.5).

The line digraph inverse of G can be computed as follows. Recall that each vertex v of G corresponds to a distinct edge of $L^{-1}(G)$. Denote this edge by $e(v)$. Note that two edges $e(v)$ and $e(w)$ enter the same vertex of $L^{-1}(G)$ iff v and w have the same successor set in G. Conversely, two edges $e(v)$ and $e(w)$ leave the same vertex of $L^{-1}(G)$ iff v and w have the same predecessor set in G. This property leads to the following construction. Let the vertex set of G be $\{v_1,\dots,v_n\}$. For each vertex v_i, find the vertex with the smallest number j such that (v_i,v_j) is an edge in G. Let $t(v_i) := j$. If there is no edge that leaves v_i, then let $t(v_i)$ be $n+1$. We observe that for all edges (v_i,v_j) that enter v_j, the number $t(v_i)$ is identical. This is because all immediate predecessors of a vertex in a VSP dag have the same set of successors. Let $s(v_j)$ denote this number $t(v_i)$. If there is no edge that enters v_j, then let $s(v_j)$ be zero. Construct a new graph G' from G as follows. The vertex set of G' consists of $n+2$ vertices, numbered from 0 to $n+1$. To avoid confusion with vertices of G, each vertex of G' is denoted by a number. For every vertex v in G, let $(s(v),t(v))$ be an edge in G'. Then, remove all vertices of G' that are isolated. Note that there may be multiple edges in G' (G' is a multidigraph), and vertex 0 is the only source of G', while vertex $n+1$ is the only sink of G'. It is not difficult to see that G' is the inverse line digraph of G, with $e(v) = (s(v),t(v))$.

We assume that G and G' are represented by incidence lists, *i.e.*, for each vertex a list of the edges incident to it is given (with outgoing *and* incoming edges). For each vertex v_i, $t(v_i)$ can be computed by a prefix-minima operation on the numbers j in the outgoing edges (v_i,v_j) of v_i. For each vertex v_j, we determine $s(v_j)$ by looking at $t(v_i)$ for one of its incoming edges (v_i,v_j). We observe the following. Let $(v_{i_1},v_j),(v_{i_2},v_j),\dots,(v_{i_k},v_j)$ be the incoming edges of v_j. Then $e(v_{i_1}), e(v_{i_2}), \ldots, e(v_{i_k})$ are the incoming edges of vertex $s(v_j)$ in G'. Conversely, let $(v_i, v_{j_1}), (v_i, v_{j_2}), \ldots, (v_i, v_{j_r})$ be the outgoing edges of v_i. Then $e(v_{j_1}), e(v_{j_2}), \ldots, e(v_{j_r})$ are the outgoing edges of vertex $t(v_i)$ in G'. Hence, to construct the list of incoming edges for vertex i in G', we take the list of incoming edges of a vertex w in G with $s(w) = i$ and replace each edge (v, w) in this list by $e(v)$. To construct the list of outgoing edges for vertex i in G', we take the list of outgoing edges of a vertex v in G with $t(v) = i$ and replace each edge (v, w) in this list by $e(w)$. All of the above can be performed using prefix operations and list ranking. We obtain:

Lemma 3.20 *The line digraph inverse of a vertex series parallel dag, consisting of e edges and n vertices, can be computed on the EREW PRAM in time $O(\log(n+e))$ using $(n+e)/\log(n+e)$ processors.*

Recently, Bodlaender and de Fluiter gave a work optimal algorithm that computes decomposition trees for ESP multidags [BdF96]. Their algorithm runs on the EREW PRAM in time $O(\log n \log^* n)$ and on the CRCW PRAM in time $O(\log n)$, where n denotes the number of edges in the multigraph. If the given VSP dag has n vertices, then its line digraph inverse has n edges and at most $n+1$ vertices. Hence, by combining the result of [BdF96] and Lemma 3.20, we can show:

Theorem 3.21 *A decomposition tree for a vertex series parallel dag with e edges and n vertices can be computed on the EREW PRAM in time $O(\log(n+e) + \log n \log^* n)$ and on the CRCW PRAM in time $O(\log(n+e))$. In either case, the number of required operations is $O(n+e)$.*

3.11.2 Computing the Heights of Defining Subgraphs

Let G be a VSP dag, and let T be a decomposition tree for G. To determine the height of the defining subgraph $G(v)$ for all vertices v of T, we apply tree contraction to T. Let each "S" vertex in T represent the function $f_S : (x,y) \mapsto x+y$, and let each "P" vertex represent the function $f_P : (x,y) \mapsto \max(x,y)$. To each leaf of T a value of 1 is assigned, and all edge functions are initially the identity. It is not difficult to see that the value of a vertex v, as defined by this expression tree, equals $\mathit{height}(G(v))$. Edge functions suitable for contraction are the functions $g_{a,b} : x \mapsto a + \max(b,x)$ (cf. Subsection 3.8.1). By Lemma 3.14, it takes $O(\log n)$ time and $n/\log n$ processors on the EREW PRAM to contract the tree. Hence, the height of $G(v)$ can be computed within the same bounds, for all vertices v simultaneously.

Theorem 3.22 *Let G be a vertex series parallel dag, and let T be a decomposition tree for G. The heights of all defining subgraphs of G with regard to T can be computed on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

3.11.3 Breadth-First Traversal

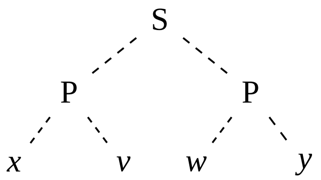
In this subsection we reduce the problem of computing a breadth-first traversal of a vertex series parallel dag to the problem of computing the breadth-first traversal of a tree. To this end, we show that for every VSP dag G and every decomposition tree T for G, we can construct a tree that essentially has the same breadth-first traversal as G with regard to T.

Definition 3.23 *Let $G = (V,E)$ be a vertex series parallel dag, and let T be a decomposition tree for G. An ordered tree $B = (W,F)$ is called a* breadth-first tree for G with regard to T *if $V \subseteq W$ and the breadth-first traversal of B restricted to V is the breadth-first traversal of G with regard to T.*

In Figure 3.7a, a VSP dag G is given, and Figure 3.7b shows a decomposition tree T for G. The breadth-first traversal of G with regard to T is 15, 16, 5, 14, 2, 12, 13, 9, 10, 11, 7, 8, 6, 3, 4, 1. The tree R in Figure 3.7d is a breadth-first tree for G with regard to T because its breadth-first traversal is r, 15, 16, 5, 14, 2, 12, 13, 9, 10, 11, 7, 8, 6, 3, 4, 1.

A breadth-first tree for G with regard to T can be determined as follows. Start with $R := G$. Add a new vertex r to R, and add an edge from r to each source of G. For each vertex w, select an edge (v,w) that enters w such that v has depth $depth(w) - 1$ in G and minimal preorder number in T. Remove all other edges that enter w. Now, R is a tree with root r. Then, for each vertex v in R, order the children of v according to their preorder number in T.

To see that R is a breadth-first tree for G with regard to T, let us first take a look at the following situation. Let (v,w) and (x,y) be edges in G that are also edges in R such that $v \neq x$, w and y have the same depth in G, and the preorder number of w in T is smaller than that of y. (Note that v and x have equal depth in G, too.) We claim that the preorder number of v in T is smaller than that of x. Assume the contrary. Since v and x have equal depth, their lowest common ancestor in T is a "P" vertex. The same holds for w and y. Clearly, the lowest common ancestor of v and w in T is an "S" vertex, and the preorder number of v in T is smaller than that of w. The same holds for x and y. We obtain the following situation in T (dashed lines indicate paths):



But in this case, (x, w) is an edge in G. Since x has a smaller preorder number in T than v, the edge (x, w) should be in R instead of (v, w). A contradiction. Hence, our assumption is wrong and the preorder number of v in T is smaller than that of x.

Our observation can also be extended to paths. Let $p = v, \ldots, w$ and $q = x, \ldots, y$ be paths in G that are also paths in R such that $v \neq x$, p and q have the same length, w and y have the same depth in G, and w has a smaller preorder number in T than y. Then v has a smaller preorder number in T than x.

Lemma 3.24 *R is a breadth-first tree for G with regard to T.*

Proof. It should be clear that the breadth-first traversal of R visits all vertices of G in the order of nondecreasing depth. Let w and y be vertices with equal depth in G such that the preorder number of w in T is smaller than that of y. We have to show that the same holds for the preorder numbers of w and y in R.

Let u be the lowest common ancestor of w and y in R, let v be the ancestor of w that is a child of u, and let x be the ancestor of y that is a child of u. As we have observed above, the preorder number of v in T is smaller than that of x because the preorder number of w in T is smaller than that of y. Recall that v and x are siblings in R and that the children of each vertex in R are ordered by their preorder number in T. As a consequence, the preorder number of v in R is smaller than that of x. We conclude that the preorder number of w in R is smaller than that of y.

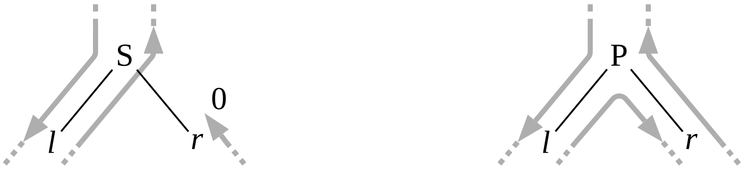
As a consequence, the breadth-first traversal of R visits the vertices of G in the order of nondecreasing depth and those vertices that have equal depth in G are visited in the order of their preorder number in T. Hence, R is a breadth-first tree for G with regard to T. □

In the following we show how R can be computed efficiently in parallel by constructing paths in the decomposition tree T. To compute R, it suffices to determine the right sibling and the parent of each vertex.

Let us first determine the right sibling of a vertex v of R. We construct a path in the decomposition tree that starts at the leaf v and moves up the tree.

If the path reaches the root or an ancestor of v that is the right child of an "S" vertex, then the path ends and the end of the path is marked with a zero. In this case, v has no right child in R, *i.e.*, v is the rightmost child of its parent in R. If the path reaches an ancestor of v that is the left child of a "P" vertex w, then the path turns round, enters the right subtree of w, and goes down the tree until it reaches a leaf. At every internal vertex u, the path continues into the left subtree of u. In the end, the path reaches a leaf x. It is not difficult to see that x and v have the same predecessors in G and hence have the same parent in R. Furthermore, among the tasks that have the same predecessors as v, x has the smallest preorder number in T that is greater than that of v. It follows that x is the right sibling of v in R.

We claim that no two paths starting at different vertices use the same tree edge in the same direction. To see this, consider how paths run along an "S" vertex and a "P" vertex:

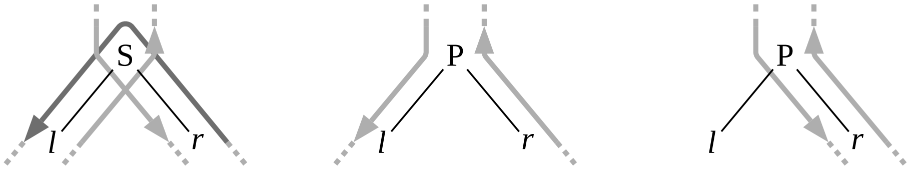


It follows that all paths can be constructed simultaneously in constant time using n processors. Then, for each leaf v in T, we only have to find the end of the path that starts at v to determine the right sibling of v or to find out that v is the rightmost child of its parent in R. To this end, we concatenate all paths into a single list and apply a prefix operation to each path using list ranking and a segmented prefix operation.

For the next step to work, it is necessary to know the height of each defining subgraph of G with regard to T (cf. previous subsection). Let v be a vertex in G that has no right sibling in R. Again, we construct a path in the decomposition tree that starts at the leaf v and moves up the tree. If the path reaches the root, then the path ends and the end of the path is marked with vertex r. In this case, the parent of v is the root of R. If the path reaches an ancestor of v that is the right child of an "S" vertex w, then the path turns round, enters the left subtree of w, and goes down until it reaches a leaf. At every "S" vertex u the path continues into the right subtree of u, while at every "P" vertex u the path continues into the subtree of u whose VSP dag is higher. If both are equally high, then the left subtree is chosen. In the end, the path reaches a leaf x. It is not difficult to see that (x, v) is an edge in G such that x has depth $\mathit{depth}(v) - 1$ in G and minimal preorder number in T. It follows that x is the parent of v

in R.

We claim that no two paths starting at leaves of T that are rightmost children in R visit any one of the tree edges in the same direction. Consider how paths run along the internal vertices of the decomposition tree:



In the middle it is shown how paths run along a "P" vertex if $height(G(l)) \geq height(G(r))$. The right figure shows the paths when the right VSP dag is higher. It is important to note that no path emerges from the left subtree of a "P" vertex w. Assume the contrary, and let v be the leaf of T at which such a path p starts. Consider the path q that we have constructed to determine the right sibling of v. This path q moves up the tree the same way as p does, until both p and q reach the "P" vertex w. Then q turns round, enters the right subtree of w, and finally ends at a leaf. It follows that v has a right sibling in R. But this contradicts the fact that we determine parents only for leaves of T that are rightmost children in R.

Hence, all paths can be constructed simultaneously in constant time on n processors. For each vertex v that has no right sibling in R, we determine the end of the path that starts at v and find the parent of v in R. As before, this is done using list ranking and a segmented prefix operation. Once each vertex knows its right sibling, and each rightmost sibling knows its parent, it is easy to determine the parents of all vertices.

By Theorem 3.22, the heights of all defining subgraphs of G with regard to T can be computed in $O(\log n)$ time using $n/\log n$ processors. Since R consists of $n+1$ vertices, the breadth-first traversal of R can be determined within the same bounds (Theorem 3.19). We have shown:

Theorem 3.25 *Given a decomposition tree T for a vertex series parallel dag G with n vertices, we can compute the breadth-first traversal of G with regard to T in time $O(\log n)$ using $n/\log n$ EREW PRAM processors.*

Figure 3.7 gives an example for the construction of a breadth-first tree. In Figure 3.7a, a vertex series parallel dag G is depicted, while Figure 3.7b (and Figure 3.7c) shows a decomposition tree T for it. The grey paths in Figure 3.7b are used to find the right siblings, while Figure 3.7c shows the paths that are

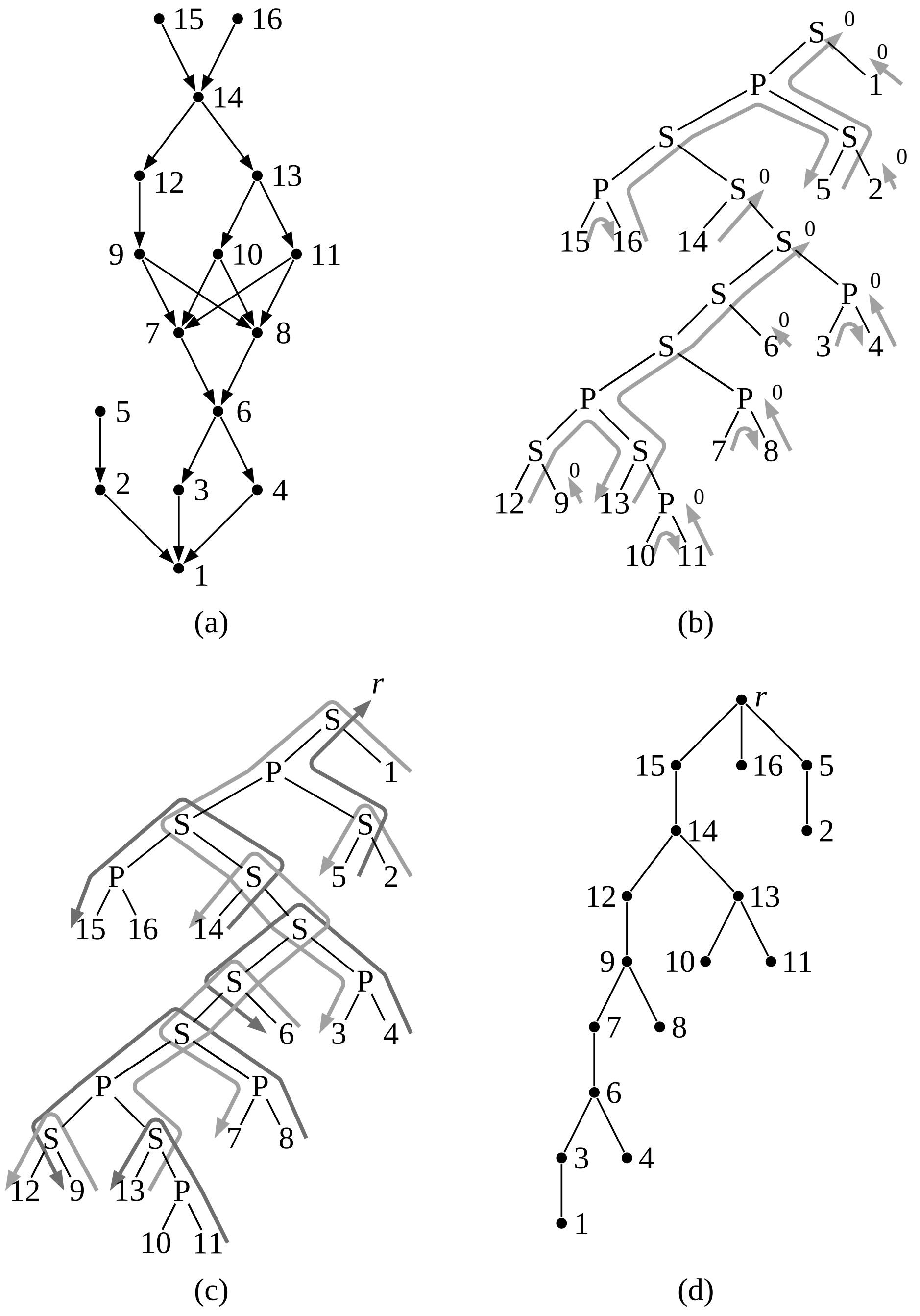


Figure 3.7: *Construction of a breadth-first tree (d) for a vertex series parallel dag (a) with regard to a decomposition tree (b,c) (see text).*

used to determine the parents for those vertices that are rightmost children in R. Finally, Figure 3.7d shows the ordered tree R defined by the right sibling and parent pointers that we have determined in Figure 3.7b and Figure 3.7c. This tree R is a breadth-first tree for G with regard to T.

3.12 Longest Paths in Directed Acyclic Graphs

Given a weighted directed acyclic graph $G = (V, E)$, the *all-pairs longest path problem* is to determine for all pairs of vertices v and w the length of a longest path from v to w in G, where the length of a path is the sum of the weights of the edges contained in the path. This problem is related to the well known problem of computing all shortest paths in G. In fact, there is a parallel algorithm for the all-pairs shortest path problem that computes longest paths instead of shortest ones after only minor modifications [DNS81, PK85]. We describe this algorithm (already adapted to longest paths) in the following.

3.12.1 Computing the Lengths

In the sequel, let x_{ij} denote the entry of matrix X in row i and column j. Let the graph G be given as an $n \times n$ weight matrix A where a_{ij} is the positive weight of the edge (i, j) if (i, j) is an edge in G and otherwise a_{ij} has the special value $-\infty$. Furthermore, $a_{ii} = 0$ for all i. (We refer to each of the n vertices of G by a number in the range $1, \ldots, n$.) We assume that $-\infty$ obeys the following rules:

$$-\infty + k = -\infty, \qquad \max(-\infty, k) = k.$$

The *max-plus matrix product* $C = A * B$ is defined by

$$c_{ij} = \max_{1 \leq k \leq n} \{a_{ik} + b_{kj}\}.$$

For $k \geq 1$, we define matrices $D^{(k)}$ as follows. If there is a path from i to j, then let $d_{ij}^{(k)}$ denote the length of the longest path from i to j that consists of at most k edges. If no path exists, let $d_{ij}^{(k)} = -\infty$. Furthermore, $d_{ii}^{(k)} = 0$ for all i. Then, $D^{(1)} = A$ and

$$d_{ij}^{(2k)} = \max_{1 \leq l \leq n} \{d_{il}^{(k)} + d_{lj}^{(k)}\}.$$

In other words, $D^{(2k)} = D^{(k)} * D^{(k)}$. Since no path in G consists of more than $n-1$ edges, every matrix $D^{(r)}$ with $r \geq n-1$ contains the lengths of all longest paths in G. Note that all matrices $D^{(r)}$ are identical for $r \geq n-1$. We call this matrix the *all-pairs longest path matrix* of G. Hence, to solve the all-pairs longest path problem we have to square A at most $\lceil \log n \rceil$ times (using the max-plus matrix product).

Let us consider the parallel complexity of this algorithm. Clearly, if a max-plus matrix product can be computed in $T(n)$ time on $P(n)$ processors, then the all-pairs longest path problem can be solved in $O(T(n) \log n)$ time on $P(n)$ processors. A max-plus matrix product $A * B$ consists of n^2 vector products $\max_{1 \leq k \leq n}\{a_{ik} + b_{kj}\}$. On the EREW PRAM each of these is computed by a prefix operation. To avoid concurrent read accesses to memory, we make $n-1$ copies of A and B beforehand. To make $n-1$ copies of one element of a matrix we require one prefix operation on an array of size n. Hence, it takes n^2 prefix operations to compute $n-1$ copies of a matrix. As a consequence, a max-plus matrix product on the EREW PRAM runs in time $O(\log n)$ and uses $n^3 / \log n$ processors. We obtain:

Theorem 3.26 *The all-pairs longest path problem for graphs with n vertices can be solved on the EREW PRAM in time* $O(\log^2 n)$ *using* $n^3 / \log n$ *processors.*

On the CRCW PRAM the time bound can be improved if all paths in the given graph have length at most $O(n)$. Under this assumption, the max-plus matrix product can be computed on the (common) CRCW PRAM in constant time using n^3 processors, since each vector product $\max_{1 \leq k \leq n}\{a_{ik} + b_{kj}\}$ takes only constant time on n processors (Theorem 3.3).

Theorem 3.27 *The all-pairs longest path problem for graphs with n vertices can be solved on the CRCW PRAM in time* $O(\log n)$ *using* n^3 *processors, provided that all paths have length* $O(n)$.

3.12.2 Computing the Transitive Closure

A problem related to the all-pairs longest path problem is the transitive closure problem on directed acyclic graphs. The *transitive closure* of a directed graph $G = (V, E)$ is the graph $G^+ = (V, E^+)$, where E^+ contains an edge (v, w) iff $v \neq w$ and there exists a path from v to w in G. If G is acyclic, then we can view G as a weighted graph with unit edge weights and apply the algorithm for the all-pairs longest path problem.

Corollary 3.28 *The transitive closure of a directed acyclic graph with n vertices can be computed on the EREW PRAM in time* $O(\log^2 n)$ *using* $n^3 / \log n$ *processors and on the CRCW PRAM in time* $O(\log n)$ *on* n^3 *processors.*

3.12.3 Extracting Longest Paths

In the beginning of this section, we have shown how to compute an all-pairs longest path matrix D, where d_{ij} is the length of a longest path between vertex i and vertex j or $-\infty$ if no path between the two vertices exists. In this subsection, we will show how to select and extract for each pair of vertices a longest path between them.

For each i and j with $d_{ij} > 0$, we select an immediate successor k of i such that d_{kj} is maximal. Let p_{ij} denote k. To select k, we require a prefix-maxima operation and hence $O(\log n)$ time on $n/\log n$ processors. Since there are n^2 pairs of vertices, we can perform all of these prefix operations in parallel using $n^3/\log n$ processors. To avoid concurrent read accesses to memory, we make $n-1$ copies of D beforehand. In the end, $i, p_{ij}, p_{p_{ij}j}, \ldots, j$ is a longest path from vertex i to vertex j if $d_{ij} > 0$.

Let $\hat{G}$ be the directed graph with vertex set $V \times V$ such that there is an edge in $\hat{G}$ from vertex (i, j) to vertex (k, j) iff $d_{ij} > 0$ and $k = p_{ij}$. Note that $\hat{G}$ is a forest of exactly n trees, where each vertex (j, j), for $1 \leq j \leq n$, is a root in $\hat{G}$. A tree with root (j, j) consists of all selected longest paths of G ending at vertex j. To extract the longest paths from $\hat{G}$ efficiently in parallel we require the trees to be ordered, *i.e.*, for each vertex an explicit order on its children must be given. To obtain an ordered representation of $\hat{G}$, we proceed as follows.

Let A be the array that consists of the elements of the set $\{(j, p_{ij}, i) | 1 \leq i, j \leq n\}$ sorted in lexicographic order. Since A consists of n^2 elements, sorting can be performed in $O(\log n)$ time using n^2 processors (Theorem 3.9). For every two successive elements (j, k, i) and (j, k, l) in A, we write l into a matrix B at position (i, j), *i.e.*, we set $b_{ij} := l$. We assume that all entries of B not written to are zero. We take (b_{ij}, j) as the right sibling of vertex (i, j), and we take (p_{ij}, j) as the parent of (i, j). If b_{ij} is zero, then vertex (i, j) is the rightmost child of (p_{ij}, j) in the order implied by A.

What remains is to extract for each leaf (i, j) in $\hat{G}$ the path from (i, j) to the root of the tree that (i, j) belongs to. To avoid concurrent read accesses to memory, we create $n-1$ copies of $\hat{G}$, resulting in n^2 trees with a total of $O(n^3)$ vertices. For every leaf (i, j) of $\hat{G}$, we choose one distinct tree $T_{(i,j)}$ among the copies of the tree that contains (i, j). This is possible, since every tree has at most n leaves. Then, for all leaves (i, j) in parallel, we extract from $T_{(i,j)}$ the path that starts at (i, j) and ends at the root of $T_{(i,j)}$. By Theorem 3.17, this takes $O(\log n)$ time on $n^3/\log n$ EREW PRAM processors. We obtain:

Theorem 3.29 *Let G be a directed weighted acyclic graph with n vertices. Selecting and extracting a longest path between vertices v and w, for all pairs of vertices v,w simultaneously, can be done on the EREW PRAM in time $O(\log n)$ using $n^3/\log n$ processors.*

3.13 Maximum Matchings in Convex Bipartite Graphs

In this section we consider the problem of computing a maximum cardinality matching in a convex bipartite graph. This problem is related to the following scheduling problem. Let (T,r,d) be a task system consisting of a set of tasks T with unit execution time and two mappings r and d that map tasks to positive integer timesteps. We call $r(x)$ the *release time* of task x and $d(x)$ the *deadline* of x. A *single processor schedule* for this task system is a mapping S from T to positive integer timesteps such that $r(x) \leq S(x)$, for all tasks x. If, in addition, $S(x) \leq d(x)$ for all $x \in T$, then we say that S *meets the deadlines*. Clearly, there exist task systems (T,r,d) where no schedule can meet all deadlines. Therefore we are interested in finding a largest possible subset U of T such that there exists a single processor schedule for (U,r,d) that meets the deadlines. Such a subset of tasks is called a *maximum feasible subset*. The problem of finding such a subset U and computing a schedule for (U,r,d) that meets the deadlines is equivalent to the problem of computing a maximum matching in a convex bipartite graph.

To see this, construct the following convex bipartite graph G from a given task system (T,r,d). Let the tasks T be one vertex set of the bipartite graph G, and let the timesteps be the other vertex set of G. Then connect each task x with all timesteps in the interval $[r(x),d(x)]$. The resulting graph is convex on the set of timesteps. Now let M be a maximum matching in this graph. Every edge (x,τ) in M pairs a task with a timestep. Let U be the subset of tasks matched in M, and let S be the schedule that maps each task of U to the timestep it is matched with in M. It is not difficult to see that S is a schedule for (U,r,d) that meets the deadlines. Moreover, no larger subset of T can have a schedule that meets the deadlines, since otherwise there would exist a matching in G that contains more edges than M. On the other hand, every convex bipartite graph can be converted into a task system (T,r,d) such that any schedule for (U,r,d) that meets the deadlines corresponds to a maximum matching in the graph, provided that U is a maximum feasible subset of T.

In [Glo67], Glover presented a simple algorithm for computing a maximum matching in a convex bipartite graph $G=(A,B,E)$. It is assumed that G is given as a list of tuples $(begin(b_1),end(b_1))$, …, $(begin(b_n),end(b_n))$, where

$B = \{b_1, \ldots, b_n\}$ and $begin(b_i)$ is the number of the the first vertex in A connected to b_i and $end(b_i)$ is the number of the last vertex in A connected to b_i (*i.e.*, the given graph is convex on A). Let L be a list of all vertices in B ordered by nondecreasing $end(\cdot)$ values. We start with an empty matching and consider one vertex of A after the other, starting with a_1. To find a match for the next vertex a_i of A, scan the list from left and pick the first vertex b with $begin(b) \leq i \leq end(b)$. Put the edge $\{a_i, b\}$ into the matching and remove b from the list. If the list is empty by now or $end(b') < i+1$, for all vertices b' in the list, then we are finished. Otherwise, we repeat the above process for vertex a_{i+1}. In the end, our matching is a maximum cardinality matching.

A parallel algorithm that computes maximum matchings in convex bipartite graphs was given by Dekel and Sahni [DS84]. Their algorithm computes matchings that are identical to the matchings obtained by Glover's strategy. In the analysis of their algorithm, Dekel and Sahni employ n processors and obtain a time bound of $O(\log^2 n)$. They used algorithms for sorting and merging that are not work optimal. However, work optimal algorithms for sorting and merging are known by now (cf. Theorems 3.5 and 3.7) and the number of processors can be reduced to $n/\log n$.

Theorem 3.30 ([DS84]) *Let $G = (A, B, E)$ be a bipartite graph that is convex on A. Let G be given as a list of tuples $(begin(b_1), end(b_1)), \ldots, (begin(b_n), end(b_n))$, where $B = \{b_1, \ldots, b_n\}$. Then a maximum matching in G can be computed on the EREW PRAM in time $O(\log^2 n)$ using $n/\log n$ processors.*

For the scheduling context, Jackson gave a strategy very similar to Glover's strategy [Jac55]. Let L be a list of all tasks ordered by nondecreasing deadline. Consider one timestep after the other, starting with timestep 1. To find a task for timestep τ, scan the list from left and pick the first task x with $r(x) \leq \tau$. Map this task to timestep τ and remove it from the list. If the list is not empty by now, repeat the above process for timestep $\tau + 1$. The schedule obtained by this algorithm is called *earliest deadline schedule* or *ED schedule*. This strategy does not find a maximum feasible subset but it finds a schedule that minimizes the maximum tardiness, *i.e.*, it minimizes the maximum number of timesteps a task is scheduled behind its deadline. Clearly, if the set T itself is a maximum feasible subset, then Jackson's rule computes a schedule for (T, r, d) that meets the deadlines. In the following we describe a fast parallel algorithm that implements Jackson's rule.

Let L be a list of all tasks ordered by nondecreasing deadline. Let us first consider a simpler problem: Instead of computing an ED schedule S from the list L, we only compute the partial timesteps of S. Given an algorithm that computes the partial timesteps, it is not difficult to come up with an algorithm that computes the actual schedule, as we will show later.

Let $r_1, \dots, r_k$ be the different release times of the tasks, and let n_i, $i = 1, \dots, k$, be the number of tasks with release time r_i. The ED schedule maps $\min(r_2 - r_1, n_1)$ tasks with release time r_1 to timesteps $r_1, \dots, r_1 + \min(r_2 - r_1, n_1) - 1$. At timestep r_2, n_2 new tasks are released and $n_1 - \min(r_2 - r_1, n_1)$ tasks with release time r_1 are still unscheduled. In general, let c_i denote the number of tasks with release time $< r_i$ that are not scheduled before timestep r_i. Then

$$\begin{aligned} c_1 &= 0 \\ c_{i+1} &= n_i + c_i - \min(r_{i+1} - r_i, n_i + c_i), \quad \text{for } i = 1, \dots, k-1. \end{aligned} \tag{3.4}$$

Given the c_i's, we can easily determine all partial timesteps in S and the length of S: Timesteps $1, \dots, r_1 - 1$ are partial, and for $i = 1, \dots, k-1$, the timesteps $r_i + \min(r_{i+1} - r_i, n_i + c_i), \dots, r_{i+1} - 1$ are partial. The last timestep occupied by a task is $r_k + n_k + c_k - 1$. Hence, it suffices to determine the c_i's. We first note that an equivalent formulation of (3.4) is

$$c_{i+1} = \max(0, c_i + n_i - (r_{i+1} - r_i)).$$

We define the following binary operator $\oplus$ on tuples of integers:

$$(a, x) \oplus (b, y) := (\max(a + y, b), x + y).$$

It is not difficult to see that $\oplus$ is associative. Let $s_1 = 0$, and let $s_i = n_{i-1} - (r_i - r_{i-1})$, for $2 \leq i \leq k$. By a simple induction on i, we can show that the following equation holds for $i = 1, \dots, k$ and suitable y_i:

$$(c_i, y_i) = \oplus_{j=1}^{i} (0, s_i).$$

As a consequence, all c_i's can be computed using a single prefix-$\oplus$ operation. We conclude that $O(\log n)$ time and $n / \log n$ processors suffice to determine the partial timesteps of S, provided that the r_i's and n_i's are given.

Let us now return to our original problem. Let L be a list of all tasks ordered by nondecreasing deadline, and let L_x be the prefix of L up to (but not including) task x. Let S be the ED schedule for L, and let S_x be the ED schedule for L_x. We observe that $S(x)$ is the earliest partial timestep in S_x later than or equal to the release time of x. This is because S restricted to the tasks of L_x is identical to S_x. We obtain the following algorithm. For each task x, we determine the partial timesteps in the schedule S_x. Then, x is mapped to the earliest partial timestep in S_x at which x is available.

In order to be able to efficiently determine the release times that occur in L_x and the number of tasks with a particular release time, it is necessary to choose L carefully. First, we sort tasks by release time such that tasks with equal release time are sorted by deadline. Let B denote the result. Let L be the list of all tasks ordered by deadline such that tasks with equal deadline appear in L in the same order as they appear in B. With regard to this list L, the list L_x consists of the tasks that have an earlier deadline than x plus the tasks that appear before x in B with the same deadline as x. Let B_x be the tasks of L_x ordered as in B. We make $n-1$ copies of B and then extract B_x from B for each x using prefix operations. Since B_x is ordered by release time, we can easily determine the different release times that occur in B_x and the number of tasks that have a certain release time. Then, for each task x, we compute the partial timesteps in the schedule S_x, using the algorithm described above. The earliest partial timestep τ_x in S_x with $r(x) \leq \tau_x$ can be computed using prefix operations. Finally, we map x to timestep τ_x. The computation of τ_x is done for all tasks simultaneously using $n^2 / \log n$ processors. The running time is $O(\log n)$.

Note that this algorithm can easily be extended to m-processor schedules. We multiply all release times by m and compute a single processor ED schedule S'. Then the schedule $S : x \mapsto \lfloor S'(x) / m \rfloor$ is an m-processor ED schedule.

Theorem 3.31 ([DUW86]) *Let (T, r, d) be a task system consisting of n unit execution time tasks with individual integer release times and deadlines. An m-processor ED schedule for (T, r, d) can be computed on the EREW PRAM in time $O(\log n)$ using $n^2 / \log n$ processors.*

In the original analysis of the algorithm given in [DUW86], n^2 processors were employed to achieve the time bound $O(\log n)$ but it is not difficult to reduce the number of processors to $n^2 / \log n$ (as we have seen).

CHAPTER 4
Scheduling Tree Orders

In this chapter we consider the problem of computing an m-processor schedule of minimal length for n unit length tasks with precedence constraints that can be represented by trees. Scheduling with tree precedence constraints 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 identical 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, for example, one outtree combined with one intree, result in intractable problems [May81, War81].

We are interested in the parallel complexity of scheduling equal length tasks constrained by a tree precedence relation. Several researchers have addressed this problem in the past. 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 reduced the problem of scheduling outtree precedence constraints to the problem of computing an earliest deadline schedule for a task system without precedence constraints but with individual task release times and deadlines [DUW86]. They gave an algorithm for this problem that runs in $O(\log n)$ time using $n^2/\log n$ processors. Alternatively, the problem of computing an earliest deadline schedule can be reduced to the problem of finding a maximum matching in a convex bipartite graph. Dekel and Sahni have shown that the latter problem can be solved on the EREW PRAM in $O(\log^2 n)$ time using $n/\log n$

processors [DS84]. See Section 3.13 for the relationship between maximum matchings in convex bipartite graphs and earliest deadline schedules.

All of these parallel algorithms have in common that they require a superlinear number of operations, which stands in contrast with the linear time sequential algorithm given in [BGJ77]. In this chapter we present a new parallel algorithm that runs on the EREW PRAM in time $O(\log n \log m)$ if $n/\log n$ processors are used. Hence, the total number of operations is only $O(n \log m)$, and if m is not part of the problem instance but a constant, then our algorithm is work and time optimal.

This chapter is organized as follows. In the next section we introduce basic concepts related to scheduling with unit length tasks, and in the section thereafter we review a sequential strategy for scheduling trees. In Section 4.3 we give basic definitions used in our algorithm. In Sections 4.4 and 4.5 we present a new class of schedules for intrees based on a particular strategy for partitioning intrees. That these schedules are actually optimal is proved in Section 4.6. In the remaining sections of this chapter we describe a parallel algorithm that computes schedules from this class.

4.1 UET Scheduling

An instance of the unit execution time (UET) scheduling problem consists of a task system $(T, \prec)$ and a number m of identical parallel target processors, where T is a set of n tasks and $\prec$ is a partial order on this set of tasks. An *m-processor (UET) schedule* for $(T, \prec)$ is a mapping S of T to positive integer timesteps such that

1. whenever $x \prec y$ then $S(x) < S(y)$ and
2. no more than m tasks are mapped to the same timestep.

The *length* or *makespan* of a schedule is the latest timestep a task is mapped to, and a schedule of minimal length is called *optimal*. Since schedules map tasks to positive integer timesteps, every optimal schedule starts at timestep 1. In terms of continuous time, timestep τ corresponds to the time interval $[\tau - 1, \tau)$.

We assume that the partial order $\prec$ is represented by a precedence graph. In Figure 4.1, such a precedence graph is depicted. The graph is an intree since all tasks have outdegree 1, except for t_{25}, which has outdegree 0. To the left of the graph the levels are shown and tasks are arranged in levels. For example, t_{12} is on a higher level than t_{16} but on the same level as t_9. Since tasks have unit length, the depth of a task in the precedence graph is the earliest possible timestep it can be scheduled. Therefore, the depth of a task x is also referred to as its *ept* value, denoted by $ept(x)$. In our example, the *ept* value (depth) of t_{15} is 2 and $ept(t_{16}) = 6$.

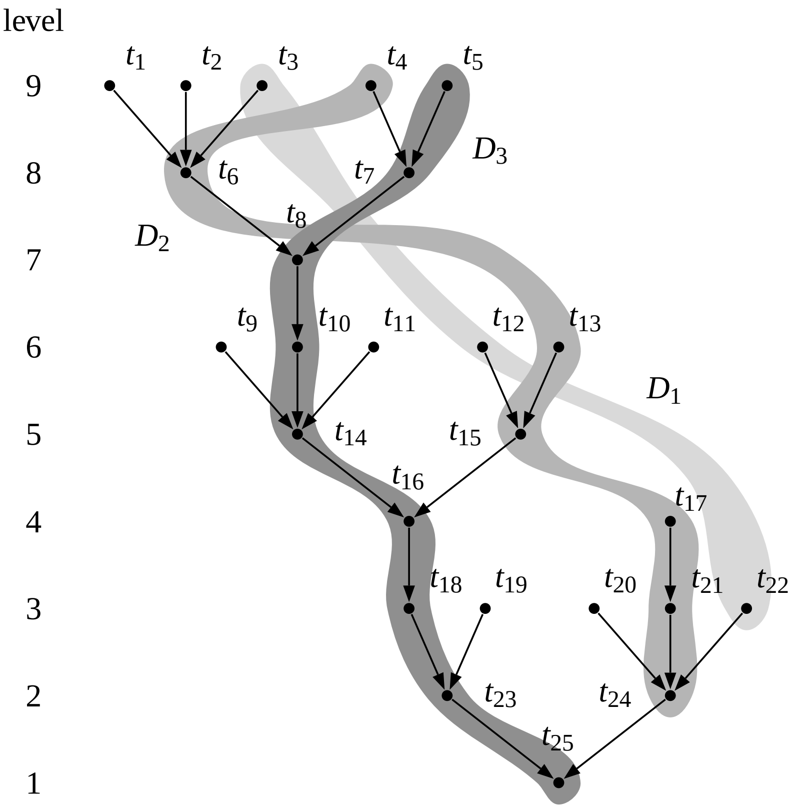


Figure 4.1: *An intree precedence graph with 25 tasks. Tasks are arranged in levels as depicted on the left. The backbones D_1, D_2, and D_3 of a possible 3-partition of the intree are shaded in three different greyscales. All remaining tasks belong to D_0. (See Section 4.4.)*

With regard to a given UET schedule, we say that x is *available* at timestep τ if all predecessors of x are mapped to timesteps earlier than τ. A timestep is *partial* if less than m tasks are mapped to it, otherwise it is called *full*. A schedule is *greedy* if at any partial timestep τ no task scheduled at a timestep later than τ is available.

A greedy 3-processor schedule for the task system given in Figure 4.1 is depicted in Figure 4.2. The length of this schedule is 10, and timesteps 8, 9, and 10 are partial.

4.2 HLF Schedules

There exists a quite simple sequential strategy for scheduling trees optimally. Schedule tasks level by level, starting with the highest level. When a level is finished and the current timestep is still partial, schedule the highest available tasks until either the current timestep is full or no more unscheduled tasks are available. The schedules obtained by this strategy are called *highest level first*.

Definition 4.1 *A schedule S is* HLF *(highest level first) if it is greedy and there do not exist tasks x and y such that $level(x) < level(y)$, $S(x) < S(y)$, and y is available at timestep $S(x)$.*

Theorem 4.2 ([Hu61, Bru82]) *Highest level first schedules are optimal for intrees and outtrees.*

A method often used in sequential scheduling algorithms is *list scheduling*. In the context of unit execution times it works as follows. We are given a list of tasks that determines for each task its priority. Assume that the first $\tau - 1$ timesteps have already been processed, that is, tasks have been mapped to them and these tasks have been removed from the list. Scan the list from left to right and pick as many of the tasks available at timestep τ as possible, up to a maximum of m. Then, remove these tasks from the list and map them to timestep τ. If there are still tasks in the list, repeat the above process for timestep $\tau + 1$. Otherwise we are finished, having obtained a schedule of length τ.

If the list consists of tasks ordered by nonincreasing level, then list scheduling produces an HLF schedule. The parallel complexity of list scheduling has been analyzed in various settings and most of the results are discouraging. For example, in [DUW86] it was shown that list scheduling is $\mathcal{P}$-complete if the precedence graph is an outtree. It is therefore unlikely that an $\mathcal{NC}$ algorithm for scheduling outtrees based on list scheduling can be found. On the other hand, fast parallel algorithms exist that compute HLF schedules for trees, albeit by rather different methods (cf. [HM87a, DUW86]). Our algorithmic approach is even more different to the sequential solution for it results in schedules that are not necessarily HLF.

In this chapter we consider intree precedence constraints only, but results apply to outtrees, outforests, and inforests as well. To schedule outtree precedences, schedule the reversed outtree with our intree algorithm and reverse the resulting schedule. To handle a forest of intrees, add one task that succeeds all roots, compute the schedule for the generated tree, and finally remove the added task from the schedule. Forests of outtrees are handled accordingly.

4.3 Ranks, Scheduling Sequences, and Rank Sequences

Let $A = \{a_1, \dots, a_k\}$ be an ordered set and let B be an arbitrary unordered set. An *ordered subset of* A is an ordered set $\{a_{i_1}, \dots, a_{i_j}\}$ such that $1 \leq i_1 < i_2 < \cdots < i_j \leq k$. By $A\langle B\rangle$ we denote the ordered subset of A that consists of the elements in the intersection of A and B.

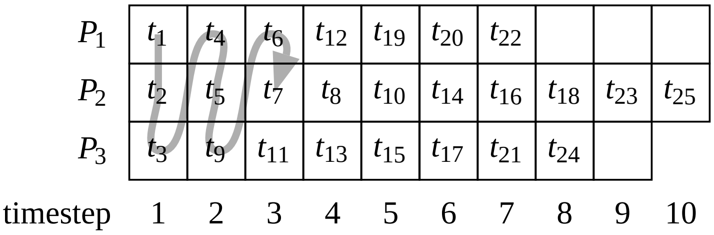
Definition 4.3 *Let A be an ordered set, let $B \subseteq A$, and let $x \in A$. Then the* rank of x in B with respect to A, *denoted by $rank_A(x : B)$, is the number of elements of B that are in front of x in A, plus 1 if $x \in B$.*

When executing a schedule, then at every partial timestep some processors have to stay idle. In order to allow formal reasoning about idle time, it is helpful to assume that every time a processor stays idle it executes an *empty task*. To give each empty task an identity, we assume that empty tasks are drawn from an ordered set $\mathcal{E} = \{e_1, e_2, \dots\}$ of sufficient cardinality.

Definition 4.4 *Let $(T, \prec)$ be a task system. A* scheduling sequence *of T is an ordered set $A = \{a_1, \dots, a_k\}$ that consists of all tasks T and all empty tasks $\mathcal{E}$ (i.e., $k = |T| + |\mathcal{E}|$) such that the i-th empty task in A is the i-th element of $\mathcal{E}$.*

Definition 4.5 *Let $(T, \prec)$ be a task system and let $A = \{a_1, \dots, a_k\}$ be a scheduling sequence of T. The* m-processor mapping of A *is the mapping $S : a_i \mapsto \lceil i/m \rceil$.*

To illustrate these concepts, let us turn to Figure 4.2. Let $\mathcal{E} = \{e_1, \dots, e_4\}$, and let A be the scheduling sequence $\{t_1, t_2, t_3, t_4, t_5, t_9, t_6, t_7, t_{11}, t_{12}, t_8, t_{13}, t_{19}, t_{10}, t_{15}, t_{20}, t_{14}, t_{17}, t_{22}, t_{16}, t_{21}, e_1, t_{18}, t_{24}, e_2, t_{23}, e_3, e_4, t_{25}\}$. The schedule depicted in Figure 4.2 is the 3-processor mapping of A and it is a schedule for the intree given in Figure 4.1. Let $B = \{t_1, t_5, t_7, t_{10}, t_{11}, t_{20}\}$. Then, $rank_A(t_{10} : \mathcal{E} \cup T) = rank_A(t_{10} : T) = 14$, $rank_A(t_{10} : B) = 5$, $rank_A(t_{23} : \mathcal{E} \cup T) = 26$, $rank_A(t_{23} : \mathcal{E} \cup B) = 8$, and $rank_A(t_{23} : B) = 6$. Now, let $C = B \cup \mathcal{E} \cup \{t_{23}\}$. Then $A\langle C\rangle = \{t_1, t_5, t_7, t_{11}, t_{10}, t_{20}, e_1, e_2, t_{23}, e_3, e_4\}$.



P_1	t_1	t_4	t_6	t_{12}	t_{19}	t_{20}	t_{22}			
P_2	t_2	t_5	t_7	t_8	t_{10}	t_{14}	t_{16}	t_{18}	t_{23}	t_{25}
P_3	t_3	t_9	t_{11}	t_{13}	t_{15}	t_{17}	t_{21}	t_{24}		
timestep	1	2	3	4	5	6	7	8	9	10

Figure 4.2: *A 3-processor schedule. The curve indicates the scheduling sequence $A = \{t_1, \ldots, e_1, t_{18}, t_{24}, e_2, t_{23}, e_3, e_4, t_{25}\}$ from which the schedule is derived. The empty tasks $e_1, \ldots, e_4$ are depicted as empty boxes.*

In our algorithm we operate on ranks of tasks in subsets of some scheduling sequence A. In order to keep track of which ranks belong to which tasks we store the rank of a task together with the task itself as a pair $(\mathit{rank}_A(x : B), x)$.

Definition 4.6 *Let A be an ordered set, and let B and C be subsets of A. Moreover, let $A\langle C\rangle = \{c_1, \ldots, c_r\}$. Then $\mathit{rank}_A(C : B)$ denotes the ordered set of pairs $\{(\mathit{rank}_A(c_1 : B), c_1), \ldots, (\mathit{rank}_A(c_r : B), c_r)\}$. We call this ordered set* rank sequence of C in B with respect to A.

Note that if $\{(u_1, x_1), \ldots, (u_r, x_r)\}$ is a rank sequence, then $u_1, \ldots, u_r$ is a nondecreasing sequence of nonnegative integers. There are a number of useful operations on rank sequences, which are introduced in the following.

Definition 4.7 *Let $U = \{(u_1, x_1), \ldots, (u_r, x_r)\}$ and $W = \{(w_1, y_1), \ldots, (w_s, y_s)\}$ be rank sequences. Then $U \bowtie W$ denotes the ordered set that is obtained by sorting the pairs of $U \cup W$ ascendingly according to the following order: $(u_i, x_i) < (u_j, x_j)$ iff $i < j$, $(w_i, y_i) < (w_j, y_j)$ iff $i < j$, and $(u_i, x_i) < (w_j, y_j)$ iff $u_i \leq w_j$.*

It is easy to see that $U \bowtie W$ can be computed by merging U and W, since the pairs of U (respectively W) appear in $U \bowtie W$ in the same order as they appear in U (respectively W). Note that $\bowtie$ is not symmetric. For example, let $U = \{(1, x_1), (4, x_2), (5, x_3)\}$ and $W = \{(2, y_1), (4, y_2), (6, y_3)\}$. Then, $U \bowtie W = \{(1, x_1), (2, y_1), (4, x_2), (4, y_2), (5, x_3), (6, y_3)\}$ but $W \bowtie U = \{(1, x_1), (2, y_1), (4, y_2), (4, x_2), (5, x_3), (6, y_3)\}$. As for an application of $\bowtie$, assume we are given $\mathit{rank}_A(X : X \cup Y)$ and $\mathit{rank}_A(Z : X \cup Y)$ for some disjoint sets $X, Y, Z \subseteq A$. Then $\mathit{rank}_A(X \cup Z : X \cup Y) = \mathit{rank}_A(X : X \cup Y) \bowtie \mathit{rank}_A(Z : X \cup Y)$. Here, the asymmetry of $\bowtie$ is necessary because if some $x \in X$ has the same rank in $X \cup Y$ as some element $z \notin X \cup Y$, then x is in front of z in A.

Definition 4.8 *Let $U = \{(u_1,x_1),\ldots,(u_r,x_r)\}$ be a rank sequence with respect to A, let $B \subseteq A$, and let C denote the ordered set $\{x_1,\ldots,x_r\}$. Also, let $\{x_{i_1}, \ldots, x_{i_k}\} = C\langle C - B\rangle$. Then $U \div B$ is the ordered set $\{(w_1,x_{i_1}),\ldots,(w_k,x_{i_k})\}$, where w_j is the number of pairs in front of (u_{i_j},x_{i_j}) in U with a second component that is contained in B, for $j = 1,\ldots,k$.*

For example, let $B = \{x_1,x_2,x_3\}$ and let U and W be as in the last example. Then $(W \bowtie U) \div B = \{(1,y_1),(1,y_2),(3,y_3)\}$. As for an application, assume that $rank_A(X \cup Z : Y)$ is given, with X and Z being disjoint. Then, $rank_A(Z : X) = rank_A(X \cup Z : Y) \div X$ and $rank_A(X : Z) = rank_A(X \cup Z : Y) \div Z$.

The next operation takes an arbitrary rank sequence and replaces the first component of each pair by the position of this pair in the rank sequence.

Definition 4.9 *Let $U = \{(u_1,x_1),\ldots,(u_r,x_r)\}$ be a rank sequence. Then the ordered set $\{(1,x_1),\ldots,(r,x_r)\}$ is denoted by $pos(U)$.*

This operation is used to construct the rank sequence of some set X in itself. For arbitrary subsets X and Y of A, it holds that $\mathrm{pos}(rank_A(X : Y)) = rank_A(X : X)$. Finally, we introduce operations that add or subtract the first components of the pairs of two rank sequences.

Definition 4.10 *Let $U = \{(u_1,x_1),\ldots,(u_r,x_r)\}$ and $W = \{(w_1,x_1),\ldots,(w_r,x_r)\}$ be rank sequences. Then*

1. *$U \oplus W := \{(u_1 + w_1,x_1),\ldots,(u_r + w_r,x_r)\}$ and*

2. *$U \ominus W := \{(u_1 - w_1,x_1),\ldots,(u_r - w_r,x_r)\}$.*

Assume we are given $rank_A(X : Y)$ and $rank_A(X : Z)$, with Y and Z being disjoint. Then, $rank_A(X : Y \cup Z) = rank_A(X : Y) \oplus rank_A(X : Z)$. Conversely, $rank_A(X : Z) = rank_A(X : Y \cup Z) \ominus rank_A(X : Y)$.

We close this section by observing that each of $\bowtie$, $\div$, pos, $\oplus$, and $\ominus$ can be performed efficiently in parallel on the EREW PRAM using merging and prefix operations. Hence, when applied to an input of size n each of $\bowtie$, $\div$, pos, $\oplus$, and $\ominus$ takes $O(\log n)$ time using $n/\log n$ processors.

4.4 Partitioning an Intree

In the sequel, let $(T, \prec)$ be a UET task system with a precedence graph that is an intree. We seek to characterize a class of scheduling sequences of T with m-processor mappings that are optimal schedules for $(T, \prec)$. Our characterization consists of three steps. First, we define a class of partitions of T, called m-partitions, that decompose the precedence tree into paths (this section). Second, we show that in a certain way every m-partition uniquely determines one scheduling sequence of T (Section 4.5). Finally, we prove that the m-processor mapping of a scheduling sequence implied by any m-partition of T is an optimal schedule for $(T, \prec)$ (Section 4.6).

Although our characterization of scheduling sequences is fairly close to being constructive, it does not provide us with an efficient parallel algorithm directly. The description of an $\mathcal{NC}$ algorithm that implements our characterization is deferred to Sections 4.7, 4.8, and 4.9.

The first step in our algorithm is to partition the task set T into $m+1$ subsets with respect to the precedence constraints. Such a partition is called *m-partition* and consists of m *backbone* sets and one set of *free* tasks. Each of the m backbones consists of one or more paths of the given intree such that the paths of one backbone have no level in common. In particular, one of the backbones is a longest path in the given intree. All tasks that are not part of any backbone are free. The idea is to let each processor execute the tasks of one backbone sequentially, *i.e.*, each backbone is "assigned" to one particular processor. Free tasks are executed by any of the m processors. It is helpful to view m-partitions as a load balancing scheme. The basic load of each processor is the backbone that is assigned to it, while free tasks are used to balance the load between processors. A formal definition of m-partitions is given next.

Definition 4.11 *An* m-partition *of* $(T, \prec)$ *consists of* $m+1$ *ordered sets* D_0, *…,* D_m *(some of them possibly empty) such that*

1. $\bigcup_{0 \le i \le m} D_i = T$,

2. $D_i \cap D_j = \emptyset$ *or* $i = j$ *for all* $0 \le i, j \le m$,

3. *each* D_i *with* $i > 0$ *contains at most one task from each level,*

4. $x \in D_i$ *implies that for all* j, $m \ge j > i$, *there exists* $y \in D_j$ *with* $level(x) = level(y)$,

5. *$x \in D_i$, $y \in D_j$, and $x \prec y$ implies $i \leq j$, and*

6. *in every D_i tasks are ordered by nonincreasing level.*

Each ordered set D_i with $i \geq 1$ is called *backbone* and the tasks therein are *backbone tasks* while the tasks in D_0 are called *free*. Let $\delta(x)$ denote the index of the subset that contains x, *i.e.*, $\delta(x) = j$ iff $x \in D_j$. Let $\gamma(x)$ denote the position of x in $D_{\delta(x)}$.

Definition 4.12 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$, let x be a backbone task, and let $\Gamma(x)$ denote the set $\{y \,|\, \delta(y) = \delta(x) - 1, level(y) = level(x)\}$. If $\Gamma(x)$ is not empty, then let $x^* \in \Gamma(x)$ be the task with $\gamma(x^*) = \max\{\gamma(y) \,|\, y \in \Gamma(x)\}$. We call x^* the* leader *of x.*

Not every task in an m-partition has a leader. In particular, no task of D_0 has a leader. Furthermore, $|\Gamma(x)| > 1$ implies $x \in D_1$ since only D_0 may contain more than one task from a certain level.

Definition 4.13 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$, let x be a backbone task, and let $\Lambda(\ell)$ denote the set of tasks on level ℓ that are leaders of some task. Then the* set of leaders *of x, denoted by $C(x)$, is the set $\{y \,|\, y \in \Lambda(level(x)), \delta(y) < \delta(x)\}$.*

Note that if $C(x)$ is not empty then $x^* \in C(x)$, and if $y \in C(x)$ and y^* exists then $y^* \in C(x)$.

Definition 4.14 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$, let x be an arbitrary task, and let $\Delta(x)$ denote the set $\{y \,|\, \delta(y) = \delta(x), level(y) > level(x)\}$. If $\Delta(x)$ is not empty, then let $x^- \in \Delta(x)$ be the task with $\gamma(x^-) = \max\{\gamma(y) \,|\, y \in \Delta(x)\}$. We call x^- the* parent *of x.*

Note that x has no parent iff x is on the highest level contained in $D_{\delta(x)}$. In particular, the first task of every D_i has no parent.

In Figure 4.1 an intree and the backbones of a possible 3-partition of it are given, with $D_3 = \{t_5, t_7, t_8, t_{10}, t_{14}, t_{16}, t_{18}, t_{23}, t_{25}\}$, $D_2 = \{t_4, t_6, t_{13}, t_{15}, t_{17}, t_{21}, t_{24}\}$, and $D_1 = \{t_3, t_{12}, t_{22}\}$. In D_0, tasks that are on the same level can be ordered arbitrarily and we fix $D_0 = \{t_1, t_2, t_9, t_{11}, t_{19}, t_{20}\}$. This 3-partition is depicted in Figure 4.3 to illustrate the concept of leaders and parents. What follows are some observations on m-partitions. By x^{*-} we denote the parent of the leader of x.

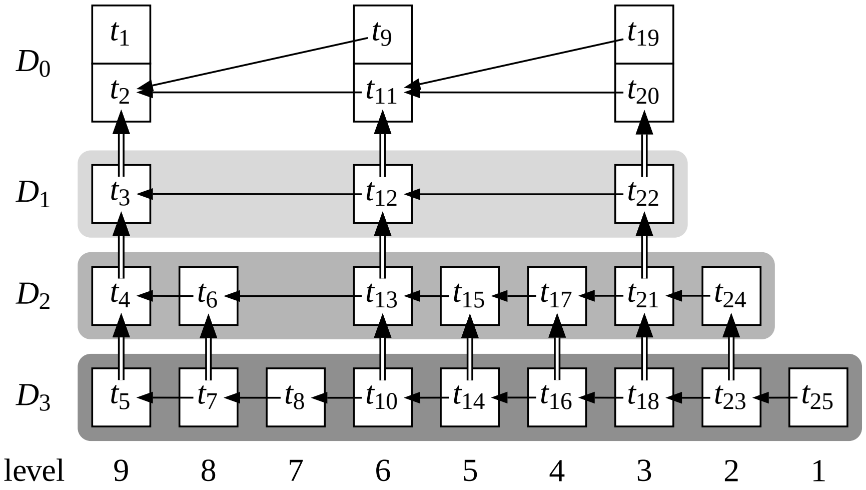


Figure 4.3: *Leaders and parents. A single line arrow points to the parent while the double line arrow points to the leader of each task. For example,* $t_{12}^{-} = t_3$ *and* $t_{22}^{*} = t_{20}$*, while* $C(t_{13}) = \{t_{12}, t_{11}\}$ *and* $C(t_{16}) = \{t_{17}\}$.

Lemma 4.15 *Let* $D_0, \ldots, D_m$ *be an m-partition of* $(T, \prec)$ *and let* $x \in T$*. Then*

1. *if x has no parent then no task in* $C(x)$ *has a parent,*
2. *if* x^- *and* x^{*-} *exist then level*$(x^{*-}) \geq$ *level*(x^-)*, and*
3. D_m *is a longest path in* $(T, \prec)$.

Proof. 1. Assume the contrary, say $y \in C(x)$ has a parent. By definition of $C(x)$, it holds that y is on the same level as x and $\delta(y) < \delta(x)$. Definition 4.11.4 implies that there exists a task $z \in D_{\delta(x)}$ that is on the same level as y^-. By definition, y^- is on a higher level than y. Therefore, z is on a higher level than x. It follows that x is not on the highest level contained in $D_{\delta(x)}$ and therefore x has a parent, a contradiction. Consequently, our assumption is wrong and no task in $C(x)$ has a parent.

2. Assume *level*$(x^{*-}) <$ *level*(x^-). Clearly, it holds that *level*$(x^*) =$ *level*(x) and *level*$(x^{*-}) >$ *level*(x^*). Therefore, *level*$(x^{*-}) >$ *level*(x). Definition 4.11.4 implies that there exists a task $z \in D_{\delta(x)}$ that is on the same level as x^{*-}. Consequently, *level*$(x^-) >$ *level*$(z) >$ *level*(x). By Definition 4.11.6, tasks in $D_{\delta(x)}$ are ordered by nonincreasing level, hence $\gamma(x^-) < \gamma(z) < \gamma(x)$. But this contradicts Definition 4.14 since both z and x^- are contained in $\Delta(x)$ and x^- is

supposed to be the task in $\Delta(x)$ with maximum γ value. We conclude that our assumption is wrong and $level(x^{*-}) \geq level(x^{-})$.

3. By Definition 4.11.5, the successor of a task of D_m is contained in D_m as well. Furthermore, at most one task from each level is contained in D_m, by Definition 4.11.3. Hence, the tasks of D_m form a path in $(T,\prec)$. That D_m is in fact a longest path follows from Definition 4.11.1 and 4.11.4. There it is implied that every task appears in some D_j and for every task x that appears in D_j with $j < m$, there exists a task in D_m that is on the same level as x. Hence, D_m contains exactly one task from every level in $(T,\prec)$. □

4.5 Partitions Imply Scheduling Sequences

In this section we show that every m-partition of $(T,\prec)$ implies one particular scheduling sequence A of T in a natural way. The implied scheduling sequence A has the following properties. If a task $x \in T$ has a parent, then the number of tasks between x and x^{-} in A is at least $m-1$. Second, if x has a leader, then x is behind x^{*} in A. Finally, A is as compact as possible, *i.e.*, the number of empty tasks in front of the last nonempty task is minimal. In the following we give a recursive characterization of this implied scheduling sequence. We first require

Definition 4.16 *Let $D_0,\ldots,D_m$ be an m-partition of $(T,\prec)$. For $-1 \leq j \leq m$ and $0 \leq i \leq m$ define $D_{i,j} := \bigcup_{i\leq k\leq j} D_k$ and $L_j := D_{0,j} \cup \mathcal{E}$.*

Note that $D_{i,j} = \varnothing$ if $i > j$, in particular, $D_{0,-1} = \varnothing$. Therefore, L_{-1} consists of empty tasks only. On the other hand, L_m consists of all tasks, including empty tasks. For the sake of convenience, let an expression of the form $rank_A(x^{*} : B)$ or $rank_A(x^{-} : B)$ equal zero if x^{*}, respectively x^{-}, does not exist.

Definition 4.17 *Let $D_0,\ldots,D_m$ be an m-partition of $(T,\prec)$ and let A be a scheduling sequence of T. Then A is* implied *by $D_0,\ldots,D_m$ iff for all $x \in D_j$ with $j \in \{0,\ldots,m\}$*

1. *$rank_A(x : D_j) = \gamma(x)$ and*

2. *$rank_A(x : L_{j-1}) = \max\left\{rank_A(x^{-} : L_{j-1}) + j - 1, rank_A(x^{*} : L_{j-1})\right\}$.*

Part 1 of the above definition ensures that tasks of every set D_j appear in A in the same order as they appear in D_j. Part 2 implies that each task x is behind its leader in A and at least $\delta(x)-1$ tasks from $L_{\delta(x)-1}$ (some of them possibly empty) are between the parent of x and x in A. In particular, $rank_A(x : L_{-1}) = 0$ for all $x \in D_0$ since no task of D_0 has a leader. Consequently, no empty tasks are in front of any task of D_0 in A.

From Definition 4.17 we can derive an algorithm for constructing a scheduling sequence that is implied by $D_0, \ldots, D_m$. Let us first note that for any disjoint subsets X, Y of A the following holds:

$$\begin{aligned} &rank_A(X \cup Y : X \cup Y) \\ &\quad = \quad (rank_A(X : Y) \oplus rank_A(X : X)) \bowtie \\ &\qquad\qquad (rank_A(Y : Y) \oplus ((rank_A(Y : Y) \bowtie rank_A(X : Y)) \div X)). \end{aligned} \tag{4.1}$$

For $j = 0, \ldots, m$, the rank sequence $rank_A(D_j : D_j)$ is given by Definition 4.17.1. If $D_j = \{x_1, \ldots, x_k\}$ then $rank_A(D_j : D_j) = \{(1, x_1), \ldots, (k, x_k)\}$. Also, $rank_A(L_{-1} : L_{-1})$ is known: $rank_A(L_{-1} : L_{-1}) = \{(1, e_1), (2, e_2), \ldots\}$ since empty tasks occur in A in the same order as in $\mathcal{E}$, by Definition 4.4. Furthermore, if $rank_A(L_{j-1} : L_{j-1})$ is known for some $j \geq 0$, then each leader of some task in D_j knows its rank (position) in L_{j-1} and we can compute $rank_A(D_j : L_{j-1})$ by applying Definition 4.17.2 to the first task of D_j, then to the second, and so on.

To construct an implied scheduling sequence, we perform the following for each j from 0 to m. First, we compute $rank_A(D_j : L_{j-1})$ using Definition 4.17.2. (For $j = 0$ and $D_0 = \{x_1, \ldots, x_k\}$ we obtain $rank_A(D_0 : L_{-1}) = \{(0, x_1), \ldots, (0, x_k)\}$, as we have observed above.) Then, we choose $X = D_j$ and $Y = L_{j-1}$ and apply equation (4.1) to compute $rank_A(L_j : L_j)$ from $rank_A(D_j : L_{j-1})$, $rank_A(D_j : D_j)$, and $rank_A(L_{j-1} : L_{j-1})$. In the end we obtain $rank_A(L_m : L_m)$, which gives us for each task of A its position in A. Note that for every m-partition of $(T, \prec)$ there exists exactly one scheduling sequence that is implied by it.

For an illustration we turn to the 3-partition depicted in Figure 4.3. Let us determine the scheduling sequence A implied by $D_0, \ldots, D_3$. Recall that $\mathcal{E} = \{e_1, \ldots, e_4\}$ in our example. It is easy to see that $A\langle L_0 \rangle$ always consists of D_0 followed by $\mathcal{E}$, hence, $A\langle L_0 \rangle = \{t_1, t_2, t_9, t_{11}, t_{19}, t_{20}, e_1, e_2, e_3, e_4\}$. Let us determine $A\langle L_1 \rangle$. The leader of t_3 is t_2, the leader of t_{12} is t_{11}, and the leader of t_{22} is t_{20}. Hence $rank_A(D_1 : L_0) = \{(2, t_3), (4, t_{12}), (6, t_{22})\}$. It follows that $A\langle L_1 \rangle = \{t_1, t_2, t_3, t_9, t_{11}, t_{12}, t_{19}, t_{20}, t_{22}, e_1, e_2, e_3, e_4\}$. Next, we determine $A\langle L_2 \rangle$. The leader of t_4 is t_3, the leader of t_{13} is t_{12}, and the leader of t_{21} is t_{22}. All other tasks of D_2 have no leader. Consequently, $rank_A(D_2 : L_1) = \{(3, t_4), (4, t_6), (6, t_{13}), (7, t_{15}), (8, t_{17}), (9, t_{21}), (10, t_{24})\}$. Then, $A\langle L_2 \rangle = \{t_1, t_2, t_3, t_4, t_9, t_6, t_{11}, t_{12}, t_{13}, t_{19}, t_{15}, t_{20}, t_{17}, t_{22}, t_{21}, e_1, t_{24}, e_2, e_3, e_4\}$. Note that there is an empty task in $A\langle L_2\rangle$ right in front of t_{24} since the rank of t_{24} in L_1 is 10, but D_0 and D_1 consist of only 9 tasks. Finally, we determine $A\langle L_3\rangle$. We compute $rank_A(D_3 : L_2) = \{(4,t_5), (6,t_7), (8,t_8), (10,t_{10}), (12,t_{14}), (14,t_{16}), (16,t_{18}), (18,t_{23}), (20,t_{25})\}$. It follows that $A\langle L_3\rangle = \{t_1, t_2, t_3, t_4, t_5, t_9, t_6, t_7, t_{11}, t_{12}, t_8, t_{13}, t_{19}, t_{10}, t_{15}, t_{20}, t_{14}, t_{17}, t_{22}, t_{16}, t_{21}, e_1, t_{18}, t_{24}, e_2, t_{23}, e_3, e_4, t_{25}\}$, which is A. The 3-processor mapping of A is shown in Figure 4.2.

4.6 Implied Scheduling Sequences are Optimal

Throughout this section let $D_0,\ldots,D_m$ be an m-partition of $(T,\prec)$ and let A be the scheduling sequence of T implied by $D_0,\ldots,D_m$. We prove in the following that the m-processor mapping of A is an optimal schedule for $(T,\prec)$. First, we establish some auxiliary propositions.

Lemma 4.18 *Let $x,y \in T$ be such that $\delta(x) < \delta(y)$ and $level(x) \geq level(y)$. Then y is behind x in A.*

Proof. Let $level(x) = level(y)$. Then either $x \in C(y)$ or $x \in D_0$ and x is not leader of a task of D_1. By Definition 4.17.2, all leaders of y are in front of y. Hence, if $x \in C(y)$ the lemma holds. Otherwise $x \in D_0$ but x is not leader of a task of D_1. By Definition 4.11.4, there exists a task in D_1 on the level of x. Therefore, there exists a task z in D_0 on the level of x that is the leader of that task in D_1 and z is behind x in D_0, by definition of leaders. It follows that z is behind x in A, since in A tasks of D_0 appear in the same order as they have in D_0, by Definition 4.17.1. Furthermore, z is in front of y, because $z \in C(y)$. Consequently, x is in front of y and the lemma holds in case $level(x) = level(y)$.

Now, let $level(x) > level(y)$. Let z be the task in $D_{\delta(y)}$ that is on the same level as x. Such a task exists according to Definition 4.11.4. Clearly, $x \in C(z)$ or $x \in D_0$ and x is not leader of a task of D_1. Hence, by the same arguments as before, z is behind x in A. The tasks in $D_{\delta(y)}$ are ordered by nonincreasing level (Definition 4.11.6) and they appear in the same order in A (Definition 4.17.1). It follows that y is behind z since z is on the same level as x and x is on a higher level than y. Consequently, y is behind x in A. □

In the following, we analyze the situation that arises if in Definition 4.17.2 the rank of some backbone task $x \in D_j$ in L_{j-1} with respect to A is determined by its leader and not by its parent.

Lemma 4.19 *Let $x \in D_j$ with $j > 0$ such that $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. Then the size of $C(x)$ is j and all tasks $C(x)$ are directly in front of x in $A\langle L_j\rangle$.*

Proof. By induction on j. Since $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$ we obtain from Definition 4.17.2 that x^* exists and $rank_A(x : L_{j-1}) = rank_A(x^* : L_{j-1})$, *i.e.*, x^* is directly in front of x in $A\langle L_j\rangle$. If $j = 1$ then $C(x) = \{x^*\}$; hence, the lemma holds for $j = 1$. Let it hold for $j - 1$ with $j - 1 > 0$, *i.e.*, our inductive hypothesis is that if $rank_A(y : L_{j-2}) > rank_A(y^- : L_{j-2}) + j - 2$ for some task $y \in D_{j-1}$, then the size of $C(y)$ is $j - 1$ and all tasks $C(y)$ are directly in front of y in $A\langle L_{j-1}\rangle$. Since $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$ there are more than $j - 1$ tasks from L_{j-1} directly in front of x in $A\langle L_j\rangle$ (and behind x^- if x^- exists). There are three cases we have to consider depending on whether x^- and/or x^{*-} exist.

case 1: x^- and x^{*-} exist. Clearly, $\delta(x^{*-}) < \delta(x^-)$ and, by Lemma 4.15, $level(x^{*-}) \geq level(x^-)$. We apply Lemma 4.18 and obtain that x^- is behind x^{*-} in A and hence in $A\langle L_j\rangle$. It follows that there are more than $j - 2$ tasks from L_{j-2} between x^{*-} and x^* in $A\langle L_{j-1}\rangle$, *i.e.*, $rank_A(x^* : L_{j-2}) > rank_A(x^{*-} : L_{j-2}) + j - 2$ (Figure 4.4). By inductive hypothesis, the tasks $C(x^*)$ are directly in front of x^* in $A\langle L_{j-1}\rangle$ and $|C(x^*)| = j - 1$. It holds that x^- is in front of all tasks $C(x^*)$ in $A\langle L_j\rangle$ since there are more than $j - 1$ tasks from L_{j-1} between x^- and x in $A\langle L_j\rangle$, x is directly behind x^* in $A\langle L_j\rangle$, and the number of tasks in $C(x^*)$ is $j - 1$. It follows that all tasks of $C(x)$ are directly in front of x in $A\langle L_j\rangle$ and $|C(x)| = j$.

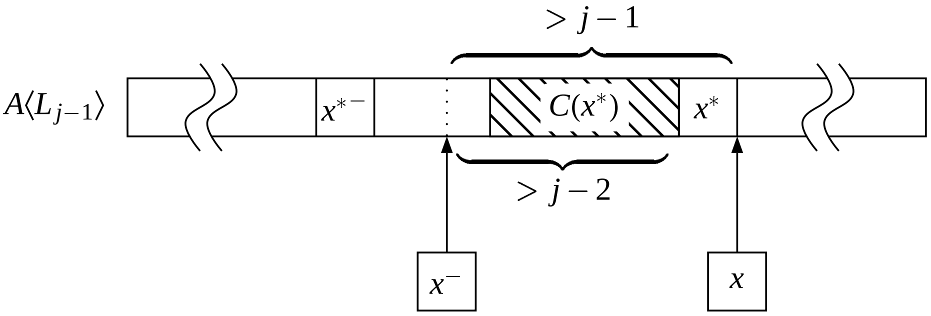


Figure 4.4: *We obtain $A\langle L_j\rangle$ by inserting tasks of D_j into $A\langle L_{j-1}\rangle$ at appropriate positions. Here we assume that the rank of $x \in D_j$ is determined by its leader and not by its parent (the arrows indicate the rank of x and x^- in L_{j-1}). Consequently, tasks $C(x)$, consisting of x^* and $C(x^*)$, are directly in front of x in $A\langle L_j\rangle$. (See proof of Lemma 4.19.)*

case 2: x^- exists but x^{*-} does not exist. Then x^* is the first task in D_{j-1}. Since there are more than $j - 1$ tasks from L_{j-1} in front of x and x is directly behind x^* in $A\langle L_j\rangle$, there are more than $j-2$ tasks from L_{j-2} in front of x^* in $A\langle L_{j-1}\rangle$, or in other words, $rank_A(x^* : L_{j-2}) > j-2$. Since $rank_A(x^{*-} : L_{j-2}) = 0$, we can apply the inductive hypothesis and obtain that tasks $C(x^*)$ are directly in front of x^* in $A\langle L_{j-1}\rangle$ and $|C(x^*)| = j-1$. As noted in the previous case, x^- is in front of all tasks $C(x^*)$ in $A\langle L_j\rangle$ and hence all tasks of $C(x)$ are directly in front of x in $A\langle L_j\rangle$. Furthermore, $|C(x)| = j$.

case 3: x^- does not exist. Then x is the first task of D_j and x^{*-} does not exist either (Lemma 4.15). Furthermore, $rank_A(x^- : L_{j-1}) = 0$ and $rank_A(x : L_{j-1}) > j-1$. Since x^* is directly in front of x in $A\langle L_j\rangle$, there are more than $j-2$ tasks from L_{j-2} in front of x^* in $A\langle L_{j-1}\rangle$, *i.e.*, $rank_A(x^* : L_{j-2}) > j-2$. By inductive hypothesis, all tasks $C(x^*)$ are directly in front of x^* in $A\langle L_{j-1}\rangle$ and $|C(x^*)| = j-1$. Therefore, all tasks of $C(x)$ are directly in front of x in $A\langle L_j\rangle$ and $|C(x)| = j$. □

Lemma 4.20 *Let $x \in D_j$ with $j > 0$ such that $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. Then there are no empty tasks in front of x in A.*

Proof. All tasks $C(x)$ are directly in front of x in $A\langle L_j\rangle$ and $|C(x)| = j$, by Lemma 4.19. Hence, $C(x)$ contains exactly one task of every D_k, $k = 0,\ldots,j-1$. Consequently, there is a task $y \in D_0 \cap C(x)$ in front of x such that there is no empty task between y and x. We have already observed that Definition 4.17.2 implies $rank_A(x : L_{-1}) = 0$ for all $x \in D_0$. Hence, empty tasks can only exist behind the last task of D_0 and therefore, no empty task is in front of y. We conclude that there is no empty task in front of x. □

Lemma 4.21 *Let $x \in D_j$ with $j > 0$ such that $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. If x^- exists then all tasks between x^- and x in $A\langle L_j\rangle$ are on the same level as x. If x^- is undefined then all tasks in front of x in $A\langle L_j\rangle$ are on the same level as x.*

Proof. Let us first note that (i) there are no empty tasks in front of x, by Lemma 4.20, (ii) all tasks $C(x)$ are on the same level as x, (iii) $C(x)$ contains exactly one task of every D_k, $k = 0,\ldots,j-1$, since the size of $C(x)$ is j, (iv) the tasks of $C(x)$ are directly in front of x in $A\langle L_j\rangle$ (Lemma 4.19), and (v) every D_k with $k = 0,\ldots,j$ is ordered by nonincreasing level and the tasks of D_k appear in this order in $A\langle L_j\rangle$, by Definition 4.11.6 and Definition 4.17.1.

It follows that all tasks in front of x in $A\langle L_j\rangle$ are on a level at least as high as x. Furthermore, if x^- exists then all tasks in front of x in $A\langle L_j\rangle$ that are higher than x are at least as high as x^-, because for every task in $D_0,\ldots,D_{j-1}$ there exists a task in D_j that is on the same level (Definition 4.11.4) and no task in

D_j is on a level between the level of x^- and x. We apply Lemma 4.18 and obtain that every task of $D_0, \ldots, D_{j-1}$ that is at least as high as x^- is in front of x^-, which proves that no task between x^- and x in $A\langle L_j\rangle$ is on a level different to that of x. On the other hand, if x^- does not exist then no task in $C(x)$ has a parent either (Lemma 4.15) and hence no task in front of x is on a level higher than x. We conclude that all tasks in front of x in $A\langle L_j\rangle$ are on the same level as x in this case. □

Lemma 4.22 *Let $x \in T$. Then there are at least $j-1$ tasks between x^- and x in $A\langle L_j\rangle$, for $j \geq \delta(x)$.*

Proof. By induction on j. According to Definition 4.17.2 there are at least $j-1$ tasks from L_{j-1} between x^- and x in $A\langle L_j\rangle$. Hence, the lemma holds for $j = \delta(x)$. Let it hold for $j-1$ with $j-1 \geq \delta(x)$, *i.e.*, our inductive hypothesis is that there are at least $j-2$ tasks between x^- and x in $A\langle L_{j-1}\rangle$. We have to show that at least one task of D_j is between x^- and x in $A\langle L_j\rangle$. Let y be the task with smallest position in D_j that is behind x in $A\langle L_j\rangle$. Such a task exists because there is a task $z \in D_j$ that is on the same level as x, by Definition 4.11.4, and task z is behind x in $A\langle L_j\rangle$, by Lemma 4.18, since $\delta(x) < \delta(z)$.

Assume that y^- does not exist. The rank of y in L_{j-1} is at least j since x and x^- are in front of y and there are at least $j-2$ tasks between x^- and x in $A\langle L_{j-1}\rangle$, by inductive hypothesis. Since $rank_A(y^- : L_{j-1}) = 0$ we derive that $rank_A(y : L_{j-1}) > rank_A(y^- : L_{j-1}) + j - 1$. By Lemma 4.21, all tasks in front of y in $A\langle L_j\rangle$ are on the same level as y. But this contradicts the fact that x and x^- are on different levels and are both in front of y. Hence, our assumption is wrong and y^- exists.

Note that, by the choice of y, we know that y^- is in front of x in $A\langle L_j\rangle$. Let $rank_A(y : L_{j-1}) = rank_A(y^- : L_{j-1}) + j - 1$, *i.e.*, there are exactly $j-1$ tasks of L_{j-1} between y^- and y in $A\langle L_j\rangle$. By inductive hypothesis, there are at least $j-2$ tasks between x^- and x in $A\langle L_{j-1}\rangle$. It follows that y^- is between x^- and x in $A\langle L_j\rangle$ (Figure 4.5). Otherwise $rank_A(y : L_{j-1}) > rank_A(y^- : L_{j-1}) + j - 1$ (Definition 4.17.2). By Lemma 4.21, all tasks between y^- and y in $A\langle L_j\rangle$ are on the same level as y. Because x^- and x are on different levels, we can derive that y^- is not in front of x^-. Furthermore, recall that y^- is not behind x. Consequently, y^- is between x^- and x in $A\langle L_j\rangle$.

By inductive hypothesis, there are at least $j-2$ tasks between x^- and x in $A\langle L_{j-1}\rangle$. We have just shown that in $A\langle L_j\rangle$ at least one task from D_j is between x^- and x. It follows that at least $j-1$ tasks are between x^- and x in $A\langle L_j\rangle$, which proves the lemma. □

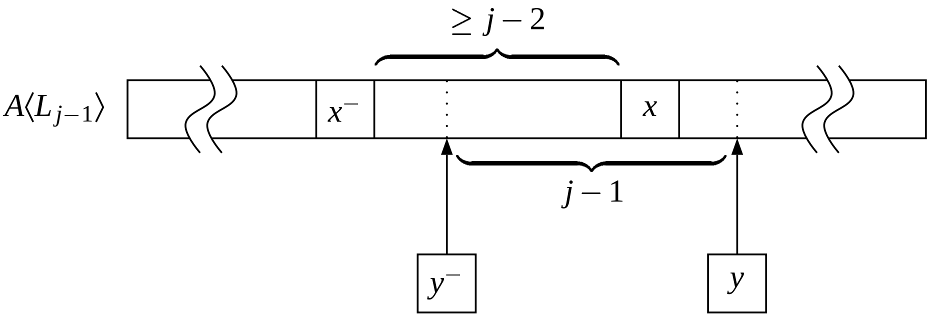


Figure 4.5: *If there are at least $j-2$ tasks between x^- and x in $A\langle L_{j-1}\rangle$ and exactly $j-1$ tasks of L_{j-1} are between y^- and y in $A\langle L_j\rangle$, then y^- is between x^- and x in $A\langle L_j\rangle$. (See proof of Lemma 4.22.)*

To prove optimality of the m-processor mapping of A, we will show that there exists a list of all tasks ordered by nonincreasing level such that the list schedule for this list has the same length as the m-processor mapping of A. To this end, we require the following:

Lemma 4.23 *Let x be a task on a longest path in $(T,\prec)$ and let $(T',\prec')$ be the forest of intrees consisting of all subtrees of $(T,\prec)$ rooted at a task on the level of x. Then there exists a list L of all tasks T' ordered by nonincreasing level such that the list schedule S for L maps x to timestep length(S).*

Proof. Let T_x denote the set of all predecessors of x plus x. Also, let L be a list of all tasks T' ordered by nonincreasing level such that for each level ℓ, the tasks of T_x on level ℓ are to the right of all other tasks of T' on level ℓ in L, *i.e.*, the tasks of T_x on level ℓ have lower priority than all other tasks on level ℓ. Let S be the list schedule for L. We prove by induction on levels (starting with the highest level) that if $y \notin T_x$, $z \in T_x$, and $level(y) = level(z)$, then $S(y) \leq S(z)$. Note that on every level in $(T',\prec')$ there is a task of T_x since x is on a longest path in $(T,\prec)$.

The claim holds for the highest level since the tasks of T_x on the highest level have lower priority in L than all other tasks on the highest level. Let the claim hold for some level ℓ and let $y \notin T_x$, $z \in T_x$, and $level(y) = level(z) = \ell - 1$. If y has no predecessor, then the list scheduling algorithm will map y to some timestep before it considers z, since z has lower priority than y in L. Otherwise, let y' be a predecessor of y such that no other predecessor of y is scheduled later than y'. Similarly, let z' be the latest predecessor of z. Clearly,

$level(y') = level(z') = \ell$. By inductive hypothesis, $S(y') \leq S(z')$. Hence, the list scheduling algorithm will map y to some timestep before it considers z, since z is available not earlier than y and y has higher priority than z, which proves the claim.

Since x is the only task in T_x on $level(x)$ we obtain that $S(x) \geq S(y)$ for all tasks y that are on the same level as x. Consequently, x is mapped to the last timestep of S. □

Now everything is in place to prove that the m-processor mapping S of the scheduling sequence implied by the m-partition is an optimal schedule. First, we show that if $x \prec y$ then at least $m-1$ tasks are between x and y in A. Consequently, y is scheduled later than x. Second, we prove that there exists a list of all tasks ordered by nonincreasing level such that the list schedule for this list has the same length as S. Since the list schedule is HLF and therefore optimal, we obtain that S is optimal too.

Theorem 4.24 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$ and let A be the scheduling sequence implied by $D_0, \ldots, D_m$. Then the mapping*

$$S : x \mapsto \lceil rank_A(x : L_m) \,/\, m \rceil$$

is an optimal m-processor schedule for $(T, \prec)$.

Proof. Let x precede y. Clearly, $level(x) > level(y)$. By Definition 4.11.5, it holds that $\delta(x) \leq \delta(y)$. If $\delta(x) < \delta(y)$ then, by Definition 4.11.4, there exists a task $z \in D_{\delta(y)}$ that is on the same level as x. Otherwise $\delta(x) = \delta(y)$ and we choose $z = x$. There exists a task $t \in D_{\delta(y)}$ that either equals y or is between z and y with $level(z) > level(t) \geq level(y)$. Note that t^- exists and either equals z or is between z and t, hence $level(z) \geq level(t^-) > level(t) \geq level(y)$. By Lemma 4.22, there are at least $m-1$ tasks between t^- and t in $A\langle L_m \rangle$. If $\delta(x) < \delta(y)$ then $\delta(x) < \delta(z)$ and, by Lemma 4.18, z is behind x. Otherwise $z = x$. In any case, there are at least $m-1$ tasks between x and y in $A\langle L_m \rangle$. We conclude that $S(x) < S(y)$. It follows that S does not violate precedence constraints and since S does not map more than m tasks to any timestep we obtain that S is a schedule for $(T, \prec)$.

It remains to show that S is optimal. The last task of D_m is the last nonempty task of $A\langle L_m \rangle$ and hence is mapped to timestep $length(S)$. Let r be the latest timestep > 1 such that there is a task $x \in D_m$ mapped to r but no task of D_m is mapped to $r-1$. If no such r exists, then to every timestep $1, \ldots, length(S)$ a task of D_m is mapped. Since D_m is a longest path in $(T, \prec)$ (Lemma 4.15) it follows that S is optimal in this case.

Otherwise there are at least m tasks directly in front of x in $A\langle L_m\rangle$ that do not belong to D_m, *i.e.*, $\mathit{rank}_A(x : L_{m-1}) > \mathit{rank}_A(x^- : L_{m-1}) + m - 1$. We now observe the following. By Lemma 4.20, there is no empty task in front of x and hence, no timestep $< r$ is partial. By Lemma 4.19, all tasks $C(x)$ are directly in front of x and the size of $C(x)$ is m, *i.e.*, $C(x)$ contains one task on the level of x from each of $D_0, \ldots, D_{m-1}$. Furthermore, tasks from each D_j, $0 \leq j \leq m$, are ordered by level in $A\langle L_m\rangle$ (Definition 4.11.6 and 4.17.1). Each backbone contains at most one task of every level (Definition 4.11.3) and in D_0 every task y that is leader of some task in D_1 has maximal position in D_0 among all tasks of D_0 on the level of y (Definition 4.12). Hence, for all $y \in C(x)$ it holds that all tasks behind y in $D_{\delta(y)}$ are on levels lower than that of y. From these observations we conclude that in $A\langle L_m\rangle$ all tasks behind x are either empty or on a level below x and all tasks in front of x are nonempty and at least as high as x.

Let $(T', \prec')$ be the forest of intrees consisting of all subtrees rooted at a task on the level of x, *i.e.*, T' consists of all tasks on the level of x and on higher levels. According to Lemma 4.23, there exists a list L' of all tasks T' ordered by nonincreasing level such that the list schedule S' for L' maps x to timestep $\mathit{length}(S')$. Let L'' be the list obtained from L' by appending all remaining tasks of T in nonincreasing level order and let S'' be the list schedule for L''. Since a list schedule for a list of tasks ordered by nonincreasing level is HLF, we obtain from Theorem 4.2 that S' is an optimal schedule for $(T', \prec')$ and S'' is an optimal schedule for $(T, \prec)$. Next, we observe that S schedules all tasks of T' in the shortest possible time, since all tasks mapped to timesteps $< r$ belong to T', no task of T' is mapped to a timestep $> r$, and no timestep earlier than r is partial. Since S' is an optimal schedule for $(T', \prec')$ we obtain $\mathit{length}(S') = S(x)$. Because $S'(x) = \mathit{length}(S')$ it follows that $S'(x) = S(x)$. A moment's reflection reveals that S'' restricted to T' is identical to S' because the list from which S' is constructed is a prefix of the list for S''. Hence, $S''(x) = S(x)$. The choice of r implies that to every timestep $r, \ldots, \mathit{length}(S)$ a task of D_m is mapped by S. Since D_m is a path in $(T, \prec)$ and both S and S'' map x to timestep r we obtain that S'' can not have a shorter length than S. Therefore, S is optimal. □

4.7 Computing an m-Partition

Our algorithm consists of three stages. In the first stage, an m-partition D_0, $\ldots$, D_m of the given intree is computed, in particular, the parent and the leader of each task is determined. In stage two, we rank tasks according to Definition 4.17, that is, we determine for $j = 0,\ldots,m$ the rank sequences $rank_A(D_j : L_{j-1})$ with respect to the scheduling sequence A implied by $D_0,\ldots,D_m$. In the last stage, we merge these rank sequences to obtain the rank sequence $rank_A(D_{0,m} : L_m)$, which gives us for each task $x \in T$ its position in A. Finally, we output the m-processor mapping of A.

We assume that the given intree is ordered, that is, for each vertex an explicit order of its immediate predecessors (children) is given. For example, the intree may be represented by a sequence of vertices, each vertex being associated with a pointer to its left sibling and a pointer to its successor.

We have shown in Sections 3.7 and 3.8 how to use the Euler-tour technique and tree contraction to compute the depth, the preorder number, and the height of each vertex of an undirected tree. When we consider the given intree precedence graph as an undirected tree (drawn with its root at the top), then it becomes clear that the depth of a task in the undirected tree corresponds to its level in the precedence graph and the height of a task in the undirected tree corresponds to its *ept* value in the precedence graph. We conclude that the level, the *ept* value, and the preorder number of each task in the precedence graph can be computed in time $O(\log n)$ using $n/\log n$ EREW PRAM processors (Theorems 3.13 and 3.15). We assume in the following that these numbers are already computed.

Lemma 4.25 *Let $(T,\prec)$ be a UET task system with a precedence graph that is an intree. Let the precedence graph be given as an ordered intree and let $|T| = n$. We can compute an m-partition of $(T,\prec)$ and determine for each task its parent and its leader in time $O(\log n \log m)$ using $n/\log n$ processors of an EREW PRAM.*

Proof. Let $>$ be an order on tasks defined as follows. We write $x > y$ or say x is greater than y if $level(x) = level(y)$ and $ept(x) > ept(y)$ or $level(x) = level(y)$ and $ept(x) = ept(y)$ and the preorder number of x in the given intree is greater than that of y. In the following we show how to determine for each task a parent and a leader such that the resulting structure is an m-partition of $(T,\prec)$.

First, we sort tasks in level order using the algorithm given in Section 3.10. This algorithm computes a breadth-first traversal of the precedence tree in time $O(\log n)$ and uses $n/\log n$ EREW PRAM processors (Theorem 3.19). Then, for each level in parallel, we determine the $m+1$ greatest tasks with respect to $>$ and sort them. Let r_ℓ denote the number of tasks on level ℓ. By Theorem 3.8, we require $O(\log r_\ell \log m)$ time to sort the $m+1$ greatest tasks on level ℓ if we use $r_\ell/\log r_\ell$ processors. We use $r_\ell/\log n$ processors and consequently require

$O(\log n \log m)$ time. To handle all levels simultaneously within the same time bound, we require $n/\log n$ processors.

Let x be the $(m-i)$-th greatest task of level ℓ, with $0 \leq i < m$. We define the $(m-i+1)$-th greatest task on level ℓ to be the leader of x. If no such task exists (because there are only $m-i$ tasks on level ℓ), then x has no leader. We define the parent of x to be the $(m-i)$-th greatest task on the next level higher than ℓ that consists of at least $m-i$ tasks. If no such level exists, then x has no parent. At this point $D_1,\ldots,D_m$ are defined and for $0 \leq i < m$ the $(m-i)$-th greatest task of each level belongs to D_{i+1}, while all other tasks belong to D_0.

Clearly, the leader of each task can be determined in constant sequential time, once the $m+1$ greatest tasks of each level are sorted. It is furthermore not difficult to construct D_m, since a task in D_m can find its parent by looking at the greatest task on the next higher level. Computing the parents of all other backbone tasks efficiently in parallel is more involved. To this end, we construct the following ordered tree R, called the *backbone tree*. The vertex set of R consists of a root vertex v and one vertex for each task in $D_1,\ldots,D_m$. The edge set of R consists of an edge (v,x) for all $x \in D_m$ and an edge (y,y^*) for every task y that has a leader. To order R, we only have to give an explicit order on the children of v, since all other vertices of R have at most one child. We order the $|D_m|$ children of v by decreasing level of their corresponding tasks, *i.e.*, the first child of v corresponds to the highest task of D_m and the last child of v corresponds to the lowest task of D_m. (Figure 4.6 shows the backbone tree R for the 3-partition given in Figure 4.3.)

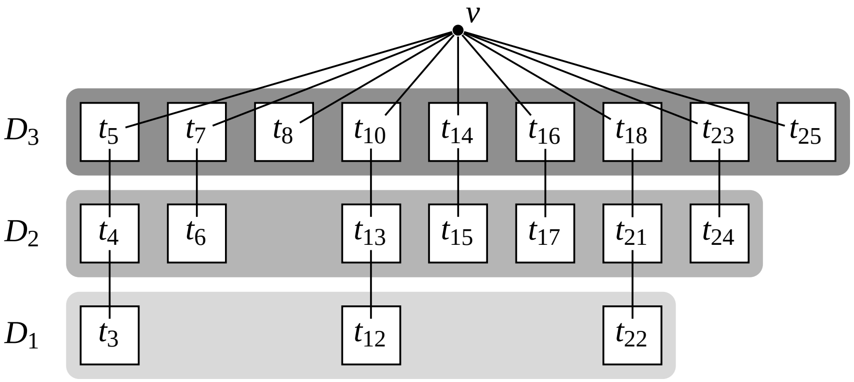


Figure 4.6: *The backbone tree (see proof of Lemma 4.25).*

Consider the breadth-first traversal L of R. It is not difficult to see that

L consists of v, followed by D_m, followed by D_{m-1}, and so on. Since the children of v are ordered by decreasing level of their corresponding tasks, the breadth-first traversal visits the tasks of each backbone in this order, *i.e.*, each backbone D_i can be found in L as a sequence of contiguous tasks ordered by decreasing level (the level of a task in the precedence graph). Consequently, to compute the backbones $D_1, \dots, D_m$ it suffices to compute the breadth-first traversal L of R. By Theorem 3.19, this can be done in $O(\log n)$ time using $n / \log n$ processors. The construction of R can be performed within the same bounds.

Let Q denote the sequence of tasks sorted in nonincreasing level order. To determine D_0 from Q, we execute the following steps. First, we remove all tasks from Q that belong to $D_1, \dots, D_m$, *i.e.*, we remove the m greatest tasks of each level, and obtain Q'. Second, we have to make sure that every leader y of some task in D_1 has highest position in D_0 among tasks on the level of y (cf. definition of leader). To this end, we perform the following in parallel for each level ℓ. Let $q_1, \dots, q_k$ be the part of Q' that consists of tasks on level ℓ. Exactly one of these tasks, say q_i, is the leader of some task in D_1, and we exchange q_i with q_k in Q'. Finally, we assign Q' to D_0.

It remains to show that the sets $D_0, \dots, D_m$ form an m-partition of $(T, \prec)$. Clearly, $D_0, \dots, D_m$ are a partition of T. Hence, properties 1 and 2 of Definition 4.11 hold. It is furthermore clear that each of $D_1, \dots, D_m$ contains at most one task from every level and each of $D_0, \dots, D_m$ is ordered by nonincreasing level. Therefore, properties 3 and 6 hold. As for property 4, we note the following. Let x be a task in D_{i+1} on level ℓ. If $i \geq 0$ then x is the $(m-i)$-th greatest task on level ℓ. It follows that there exist at least $m-i$ tasks on level ℓ. Consequently, there is a task from level ℓ in each of $D_{i+2}, \dots, D_m$. Otherwise $x \in D_0$. We removed the m greatest tasks of level ℓ to obtain the tasks for D_0. Hence, there are at least $m+1$ tasks on level ℓ and each of $D_1, \dots, D_m$ contains one of them.

Now let $x \in D_i$ and $y \in D_j$ with $x \prec y$. Then there exist $m-j$ tasks on the level of y that are greater than y (there may exist even more if $y \in D_0$). By definition of $>$, every task on the level of y that is greater than y has either greater *ept* value or the same *ept* value and greater preorder number than y. We claim that for every task z on the level of y that is greater than y, there exists a distinct task z' on the level of x that is greater than x. To prove the claim, let z be a task on the level of y with $z > y$ and consider the following two cases.

case 1: $ept(z) > ept(y)$. Then the longest path of tasks in $(T, \prec)$ preceding z is longer than the longest path of tasks that precede y. Hence, there exists a predecessor z' of z that is on the same level as x and $ept(z') > ept(x)$. Therefore $z' > x$.

case 2: $ept(z) = ept(y)$, but z has a greater preorder number than y. Then the longest path of tasks in $(T, \prec)$ preceding z has the same length as the longest path of tasks that precede y. It follows that there exists a predecessor z' of z on the level of x with $ept(z') \geq ept(x)$. We observe that every predecessor of z has a greater preorder number than any predecessor of y, because z has a greater preorder number than y. Consequently, the preorder number of z' is greater than that of x. Hence $z' > x$.

Because $(T, \prec)$ forms an intree, we automatically choose in the above cases for every z a distinct z', which proves the claim. As noted before, there exist at least $m - j$ tasks on the level of y that are greater than y. By the above claim, there exist at least $m - j$ tasks on the level of x that are greater than x. Since $x \in D_i$ we obtain that $i \leq j$, which proves property 5 of Definition 4.11.

We conclude that $D_0, \ldots, D_m$ obtained by way of the above algorithm conforms to Definition 4.11 and hence is an m-partition of $(T, \prec)$. □

4.8 Ranking Tasks

The characterization of implied scheduling sequences given in Definition 4.17 is recursive in the number of processors m. As we have seen, this definition is useful for proving that the m-processor mapping of the implied scheduling sequence is an optimal schedule. On the other hand, it is not clear how a fast parallel algorithm can be derived from it directly. In this section we give a version of Definition 4.17.2 that is more open to parallelization. In the sequel let $\alpha(x)$ denote $\sum_{y \in C(x)} \gamma(y)$.

Lemma 4.26 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$ and let A be the scheduling sequence implied by $D_0, \ldots, D_m$. Then for any $x \in D_j$ with $j \geq 1$*

$$rank_A(x : L_{j-1}) \geq \alpha(x).$$

Proof. For every task y it holds that if y has a leader, then y is behind its leader in A (Definition 4.17.2). Consequently, x is behind all tasks of $C(x)$. By Definition 4.17.1, the tasks of every D_i appear in the same order in A as they appear in D_i. Hence, every $y \in C(x)$ is behind $\gamma(y) - 1$ tasks from $D_{\delta(y)}$ in A. We conclude that at least $\alpha(x)$ tasks of L_{j-1} are in front of x in A. □

Lemma 4.27 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$ and let A be the scheduling sequence implied by $D_0, \ldots, D_m$. Let $x \in D_j$ with $j \geq 1$ such that $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. Then*

$$rank_A(x : L_{j-1}) = \alpha(x).$$

Proof. We can use the same arguments as in the proof of Lemma 4.26. Additionally, we observe that the tasks of $C(x)$ are directly in front of x in $A\langle L_j \rangle$ and the size of $C(x)$ is j (Lemma 4.19), hence, $C(x)$ contains exactly one task of each set $D_0, \ldots, D_{j-1}$. Furthermore, no empty tasks are in front of x in A, by Lemma 4.20. Consequently, there are exactly $\alpha(x)$ tasks of L_{j-1} in front of x in A. □

The above two lemmas lead to a nonrecursive expression for the rank of a backbone task $x \in D_j$ in L_{j-1}, which is given next.

Lemma 4.28 *Let $D_0, \ldots, D_m$ be an m-partition of $(T, \prec)$ and let A be the scheduling sequence implied by $D_0, \ldots, D_m$. Also, let $x \in D_j$ with $j \geq 1$. Then*

$$\begin{aligned} & rank_A(x : L_{j-1}) \\ = \quad & \max\left\{ \gamma(x)(j-1), \max_{y \in D_j, \gamma(y) \leq \gamma(x)} \{\alpha(y) + (\gamma(x) - \gamma(y))(j-1)\} \right\}. \end{aligned}$$

Proof. Let us first state that

$$rank_A(x : L_{j-1}) = \max\left\{ rank_A(x^- : L_{j-1}) + j - 1, \alpha(x) \right\}. \tag{4.2}$$

To prove this, let $\alpha(x) > rank_A(x^- : L_{j-1}) + j - 1$. By Lemma 4.26, it holds that $rank_A(x : L_{j-1}) \geq \alpha(x)$ and therefore $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. Consequently, $rank_A(x : L_{j-1}) = \alpha(x)$, by Lemma 4.27. Otherwise $rank_A(x^- : L_{j-1}) + j - 1 \geq \alpha(x)$. Assume that $rank_A(x : L_{j-1}) > rank_A(x^- : L_{j-1}) + j - 1$. By Lemma 4.27, we obtain $rank_A(x : L_{j-1}) = \alpha(x)$ and hence $\alpha(x) > rank_A(x^- : L_{j-1}) + j - 1$, a contradiction. It follows that our assumption is wrong and $rank_A(x : L_{j-1}) \leq rank_A(x^- : L_{j-1}) + j - 1$. On the other hand, $rank_A(x : L_{j-1})$ is at least $rank_A(x^- : L_{j-1}) + j - 1$, by Definition 4.17.2. Therefore, $rank_A(x : L_{j-1}) = rank_A(x^- : L_{j-1}) + j - 1$, which proves equation (4.2).

We prove the lemma by induction on $\gamma(x)$. If $\gamma(x) = 1$ then $rank_A(x^- : L_{j-1}) = 0$ since x^- is undefined, and we obtain

$$\max\left\{ \gamma(x)(j-1), \max_{y \in D_j, \gamma(y) \leq \gamma(x)} \{\alpha(y) + (\gamma(x) - \gamma(y))(j-1)\} \right\}$$

$$\begin{aligned}
&= \max\{j-1,\alpha(x)\} \\
&\overset{(4.2)}{=} rank_A(x : L_{j-1}).
\end{aligned}$$

It follows that the lemma holds for $\gamma(x)=1$. Let the lemma hold for the parent of x. Using the fact that $\gamma(x)=\gamma(x^-)+1$ and using the inductive hypothesis, we derive

$$\begin{aligned}
& \max\left\{\gamma(x)(j-1), \max_{y\in D_j,\gamma(y)\le\gamma(x)}\{\alpha(y)+(\gamma(x)-\gamma(y))(j-1)\}\right\} \\
= {} & \max\left\{\gamma(x)(j-1), \max_{y\in D_j,\gamma(y)\le\gamma(x^-)}\{\alpha(y)+(\gamma(x)-\gamma(y))(j-1)\},\alpha(x)\right\} \\
= {} & \max\left\{\begin{array}{l}\gamma(x^-)(j-1)+j-1, \\ \max\limits_{y\in D_j,\gamma(y)\le\gamma(x^-)}\big\{\alpha(y)+(\gamma(x^-)-\gamma(y))(j-1)\big\}+j-1, \\ \alpha(x)\end{array}\right\} \\
\overset{\text{i.h.}}{=} {} & \max\big\{rank_A(x^- : L_{j-1})+j-1,\alpha(x)\big\} \\
\overset{(4.2)}{=} {} & rank_A(x : L_{j-1}).
\end{aligned}$$

□

Using Lemma 4.28, it is now easy to rank all tasks efficiently in parallel.

Lemma 4.29 *Given for each task its leader and its parent in an m-partition $D_0,\ldots,D_m$ of $(T,\prec)$, we can compute the rank sequences $rank_A(D_0 : L_{-1})$, …, $rank_A(D_m : L_{m-1})$ in time $O(\log n)$ using $n/\log n$ processors of an EREW PRAM, where A is the scheduling sequence implied by $D_0,\ldots,D_m$ and $|T|=n$.*

Proof. For each task x in parallel, we determine $\gamma(x)$, $\delta(x)$, and $\alpha(x)$ using list ranking and segmented prefix-sums operations on the lists of leaders and parents. Next, we compute $rank_A(x : L_{\delta(x)-1})$, as proposed in Lemma 4.28, using a prefix-maxima operation on each of $D_1,\ldots,D_m$. It is now easy to construct the ordered sets $rank_A(D_j : L_{j-1})$ for $j=1,\ldots,m$. For $j=0$, observe that if $D_0=\{x_1,\ldots,x_k\}$ then $rank_A(D_0 : L_{-1})=\{(0,x_1),\ldots,(0,x_k)\}$ since $L_{-1}=\mathcal{E}$ and in A no empty tasks are in front of any task of D_0 (Definition 4.17.2). Clearly, all of the above can be performed in time $O(\log n)$ on $n/\log n$ EREW PRAM processors (Theorems 3.2 and 3.11). □

4.9 Constructing the Implied Scheduling Sequence

We aim to determine $\mathit{rank}_A(D_{0,m} : L_m)$, which contains for each nonempty task its position in the scheduling sequence implied by $D_0, \ldots, D_m$. In the following we show how to compute $\mathit{rank}_A(D_{0,m} : L_m)$ using a binary tree computation, an outline of which is given next.

We assume without loss of generality that $m+1$ is a power of 2. Our computation tree is a complete binary tree with $m+1$ leaves numbered from 0 to m. Leaf number j carries the rank sequence $\mathit{rank}_A(D_j : L_j)$. Each internal vertex of the binary tree represents an operation that takes the two rank sequences of its two children as input and outputs a new rank sequence. The rank sequence that is output by the root vertex of the computation tree will be $\mathit{rank}_A(D_{0,m} : L_m)$. The operation associated with each internal vertex of the computation tree consists of a number of applications of $\bowtie$, $\div$, pos, $\oplus$, and $\ominus$ and is presented in the next lemma.

Lemma 4.30 *Let A be an ordered set and let X, Y, and Z be disjoint subsets of A. Given the rank sequences $\mathit{rank}_A(Y : X \cup Y)$ and $\mathit{rank}_A(Z : X \cup Y \cup Z)$, we can compute $\mathit{rank}_A(Y \cup Z : X \cup Y \cup Z)$ in time $O(\log(|Y|+|Z|))$ using $(|Y|+|Z|)/\log(|Y|+|Z|)$ processors of an EREW PRAM.*

Proof. Clearly, $\mathrm{pos}(\mathit{rank}_A(Z : X \cup Y \cup Z)) = \mathit{rank}_A(Z : Z)$. Hence,

$$\mathit{rank}_A(Z : X \cup Y) = \mathit{rank}_A(Z : X \cup Y \cup Z) \ominus \mathrm{pos}(\mathit{rank}_A(Z : X \cup Y \cup Z)).$$

Then, we determine $\mathit{rank}_A(Y \cup Z : X \cup Y)$ using a relationship observed in section 4.3, namely

$$\mathit{rank}_A(Y \cup Z : X \cup Y) = \mathit{rank}_A(Y : X \cup Y) \bowtie \mathit{rank}_A(Z : X \cup Y).$$

Finally, we compute

$$\begin{aligned} &\mathit{rank}_A(Y \cup Z : X \cup Y \cup Z) = \\ &\qquad (\mathit{rank}_A(Y : X \cup Y) \oplus (\mathit{rank}_A(Y \cup Z : X \cup Y) \div Z)) \bowtie \mathit{rank}_A(Z : X \cup Y \cup Z). \end{aligned}$$

As noted in section 4.3, we require $O(\log n)$ time on $n/\log n$ EREW PRAM processors to perform any of $\bowtie$, $\div$, pos, $\oplus$, and $\ominus$ on inputs of size n. Hence, $\mathit{rank}_A(Y \cup Z : X \cup Y \cup Z)$ can be computed within the desired resource bounds. □

The height of our binary computation tree is logarithmic in m since the tree is complete and there are $m+1$ leaves. We divide the execution of the computation tree in phases. In each phase we process all vertices of one level in parallel. After $O(\log m)$ such phases the output of the root vertex is known. As we will see in the next lemma, we require $O(\log n)$ time using $n/\log n$ processors to perform one phase. Therefore, we can compute $rank_A(D_{0,m} : L_m)$ in time $O(\log n \log m)$ if $n/\log n$ processors are available. Details on this binary tree computation are given next.

Lemma 4.31 *Let $D_0,\dots,D_m$ be an m-partition of $(T,\prec)$ and let $|T| = n$. Given the rank sequences $rank_A(D_i : L_i)$, for $i = 0,\dots,m$, we can compute the rank sequence $rank_A(D_{0,m} : L_m)$ in time $O(\log n \log m)$ using $n/\log n$ processors of an EREW PRAM.*

Proof. Without loss of generality, we can assume that $m+1$ is a power of two. For $j = 1,\dots,\log(m+1)$, let $I(j)$ be the set of integer intervals that partition the interval $[0,m]$ into closed integer intervals each of size 2^j, *e.g.*, $I(1) = \{[0,1],[2,3],\dots,[m-1,m]\}$, $I(2) = \{[0,3],[4,7],\dots,[m-3,m]\}$, and $I(\log(m+1)) = \{[0,m]\}$.

We start with the given $m+1$ rank sequences $rank_A(D_{i,i} : L_i)$ as input. For $j = 1, \dots, \log(m+1)$ we perform the following for each interval $[a,c] \in I(j)$ in parallel. Let $b = a+(c-a+1)/2$. We apply the algorithm given in the proof of Lemma 4.30 to the rank sequences $rank_A(D_{a,b-1} : L_{b-1})$ and $rank_A(D_{b,c} : L_c)$ to obtain the rank sequence $rank_A(D_{a,c} : L_c)$. (In order to apply Lemma 4.30, let L_{a-1} correspond to X, let $D_{a,b-1}$ correspond to Y, and let $D_{b,c}$ correspond to Z. Then $L_{b-1} = X \cup Y$, $L_c = X \cup Y \cup Z$, and $D_{a,c} = Y \cup Z$.) Using $|D_{a,c}|/\log n$ processors, we require $O(\log n)$ time. Since $\sum_{[a,c]\in I(j)} |D_{a,c}| = n$, we can handle all intervals $I(j)$ simultaneously using $n/\log n$ processors within the same time bound.

After $\log(m+1)$ iterations, we obtain the rank sequence $rank_A(D_{0,m} : L_m)$, which is the desired result. The time required to perform all iterations is $O(\log n \log m)$ and $n/\log n$ processors are used. □

We are now ready to state the main result of this chapter.

Theorem 4.32 *Let $(T,\prec)$ be a UET task system with a precedence graph that is an intree. Let the precedence graph be given as an ordered intree, and let $|T| = n$. We can compute an optimal m-processor schedule for $(T,\prec)$ on the EREW PRAM in $O(\log n \log m)$ time using $n/\log n$ processors.*

Proof. In the first step, we determine an m-partition $D_0, \ldots, D_m$ of $(T, \prec)$. By Lemma 4.25, this can be performed in time $O(\log n \log m)$ using $n / \log n$ processors. Next, we compute the rank sequences $\mathit{rank}_A(D_j : L_{j-1})$ for all $j \in \{0, \ldots, m\}$, where A is the scheduling sequence of T implied by $D_0, \ldots, D_m$. According to Lemma 4.29, we require $O(\log n)$ time on $n / \log n$ processors. Then, we compute the rank sequences $\mathit{rank}_A(D_j : L_j)$, for $j = 0, \ldots, m$. This is easy, because we obtain $\mathit{rank}_A(x : L_{\delta(x)})$ from $\mathit{rank}_A(x : L_{\delta(x)-1})$ by adding $\gamma(x)$. We process these rank sequences using the algorithm given in Lemma 4.31 and obtain $\mathit{rank}_A(D_{0,m} : L_m)$. Finally, we map each task x of T to timestep $\lceil \mathit{rank}_A(x : L_m) / m \rceil$. By Theorem 4.24, this mapping is an optimal schedule for $(T, \prec)$. □

CHAPTER 5

The Two Processor Scheduling Problem

The two processor scheduling problem is interesting for various reasons. First, it lies close to the border between intractable scheduling problems and those for which efficient algorithms are known. This border is a challenging area for investigations. If the number of processors for which we compute the schedule is not limited to two but is part of the problem instance, then the problem becomes $\mathcal{NP}$-hard, as was shown by Ullman [Ull75]. On the other hand, it is unknown whether polynomial algorithms exist that compute optimal k-processor schedules for any fixed k greater than two. Second, optimal two processor schedules can be used to compute maximum matchings in complements of comparability graphs. Results on the complexity of it might shed new light on the complexity of the maximum matching problem in general graphs.

It is therefore not surprising that the two processor scheduling problem has a long history, starting in 1969 when Fujii, Kasami, and Ninomiya proposed the first polynomial algorithm. It is based on the idea to construct an optimal two processor schedule from a maximum matching in the incomparability graph of the given partial order [FKN69]. Later, Coffman and Graham found an $O(n^2)$ algorithm based on list scheduling, where the sequence of tasks in the list is determined by a lexicographic numbering scheme [CG72]. Their algorithm requires the given precedence graph to be either transitively closed or transitively reduced. Sethi showed that Coffman and Graham's algorithm can be implemented to run in time $O(n\alpha(n)+e)$ [Set76], where e is the number of edges in the precedence graph and $\alpha(n)$ is an inverse of Ackermann's function. Finally, Gabow developed an algorithm that runs within the same time bound but does not require the precedence graph to be transitively closed or reduced [Gab82]. The results of Sethi and Gabow can be combined with a result on static union-find given in [GT85] to obtain algorithms that run in time $O(n+e)$.

The parallel complexity of the two processor scheduling problem was first investigated by Vazirani and Vazirani [VV85]. They gave an algorithm based on a randomized algorithm for maximum matchings with an expected running time that is a polynomial in the logarithm of the number of tasks. The first $\mathcal{NC}$ algorithm was developed by Helmbold and Mayr [HM87b]. It consists, roughly speaking, of two components. The *distance* algorithm computes the length of an optimal two processor schedule for some task system, while the remaining algorithm uses the distance algorithm to construct the actual schedule. The distance algorithm runs in time $O(\log^2 n)$ using n^5 processors and the total requirements are $O(\log^2 n)$ time and n^{10} processors.

Since then, a number of attempts have been made to develop more efficient parallel algorithms. Moitra and Johnson [MJ89] and H. Jung, Serna, and Spirakis [JSS91] proposed a new distance algorithm but unfortunately their algorithm is wrong, as a counterexample given in [Jun92] shows. N. Jung proposed a different distance algorithm that requires n^4 processors but its proof of correctness is not beyond doubt. However, combining the original distance algorithm of Helmbold and Mayr with the second component of N. Jung's algorithm [Jun92], one obtains a two processor scheduling algorithm that runs in time $O(\log^2 n)$ using n^5 processors.

In this chapter we present a new parallel algorithm for the two processor scheduling problem. It requires $O(\log^2 n)$ time and only $n^3/\log n$ CREW PRAM processors. Our main contribution is a novel and efficient distance algorithm. To compute the actual schedule using information obtained from the distance algorithm, we mainly follow N. Jung. Our contributions to this part of the algorithm are simplifications and a proof of correctness that is more rigorous than that given in [Jun92].

The rest of this chapter is organized as follows. In Sections 5.1 and 5.2 we introduce notation and basic concepts related to the two processor scheduling problem. In the section thereafter we show how knowledge of the length of an optimal two processor schedule for a task system can be used to gain information on the structure of an optimal schedule. The following three sections are dedicated to the distance algorithm and its proof of correctness. In Section 5.7

we use maximum matchings in convex bipartite graphs to determine the structure of an optimal schedule, while in Section 5.8 the final schedule is computed. We close this chapter in Section 5.9, where we apply our two processor scheduling algorithm to the problem of computing maximum matchings in co-comparability graphs.

5.1 LMJ Schedules

Let $(T, \prec)$ be an arbitrary task system. We assume that $\prec$ is given as a precedence graph. We wish to find a two processor UET schedule for $(T, \prec)$ of minimal length (cf. Section 4.1). Figure 5.1 shows a precedence graph of a task system with 15 tasks. An optimal two processor UET schedule for it is depicted in Figure 5.1. A well known scheduling strategy is to schedule tasks on higher levels earlier than others whenever possible. If the precedence constraints are trees, then this simple strategy is sufficient (cf. Section 4.2). In the two processor case with arbitrary precedence constraints, this strategy must be refined.

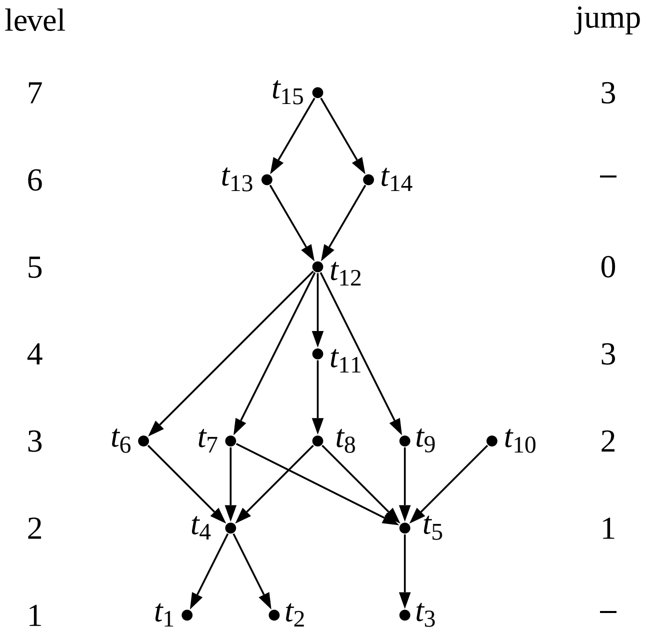
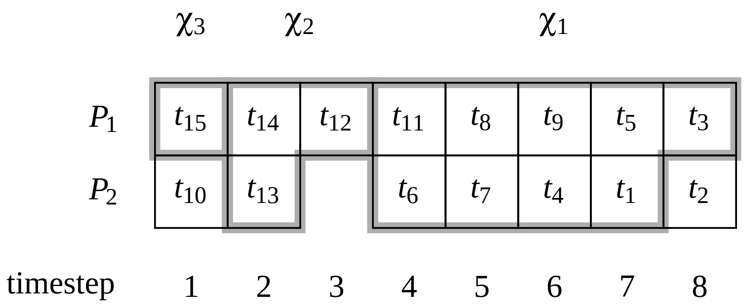


Figure 5.1: *A precedence graph with 15 tasks. On the left are the levels and on the right is the jump sequence of an LMJ schedule (3,0,3,2,1).*



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

Figure 5.2: *An LMJ schedule for the above precedence graph. The blocks of a possible block decomposition* χ_3, χ_2, χ_1 *are framed in grey (see Section 5.5).*

Assume that tasks on levels $L, \ldots, \ell+1$ have already been mapped to timesteps and there are k unmapped tasks remaining on level ℓ. We map those tasks to the next $\lceil \frac{k}{2} \rceil$ timesteps. If k is odd, then only one task t of level ℓ is mapped to the last of the $\lceil \frac{k}{2} \rceil$ timesteps and we try to pair t with a task t' from a lower level ℓ'. In this case we say that level ℓ *jumps* to level ℓ' and we call t' a *fill-in* task. If there is no task available that t can be paired with, then t is paired with an empty task and we say that ℓ jumps to level 0. If a level ℓ jumps to level $\ell - 1$ or 0, then we say that this jump is *trivial*. Schedules that can be constructed by iterating the above process are called *level schedules* and the sequence of levels jumped to, in decreasing order of levels jumped from, is called the *jump sequence* of a level schedule. A jump sequence that is lexicographically greater than any other jump sequence of a level schedule for $(T, \prec)$ is called a *lexicographically maximum jump* (LMJ) sequence and a level schedule with an LMJ sequence is called an *LMJ schedule*.

Theorem 5.1 ([Gab82]) *Every LMJ schedule is optimal.*

In the following we outline how LMJ schedules can be computed in parallel. First, we determine which levels jump. To accomplish this, we augment the precedence graph by some additional tasks and compute for each pair of tasks t and t' in parallel the length of an optimal two processor schedule for the tasks that are both successors of t and predecessors of t'. By comparing these “scheduling distances” for suitable pairs of tasks, we are able to determine which levels jump. Then, we determine the LMJ sequence. We use the precedence graph and the levels that jump to construct a bipartite graph. One set of vertices in this graph represents levels that jump and the other set represents tasks that are possible candidates for fill-ins. Since the resulting graph is convex we can compute a special kind of maximum matching from which we obtain the LMJ sequence. Finally, we determine the tasks used in the jumps of an LMJ schedule. To this end, an "implication graph" is constructed that reflects the dependencies between fill-in tasks and tasks that can be used to jump from levels. This is to ensure that no task is used as a fill-in if it is required to jump from a level. It suffices to determine the transitive closure of the implication graph to obtain the pairs of tasks that can be used for the jumps. Once the tasks used in jumps are determined it is easy to compute the actual schedule.

5.2 Notation

In the following we introduce some notation used throughout the rest of this chapter. For this purpose, let t, t', and x be tasks of some task system $(T,\prec)$ and let A be a subset of T. With regard to a given schedule, a pair of tasks (t,t') from different levels is called an *actual jump* if t and t' are mapped to the same timestep. Note that t' may be an empty task here. By $I(t,t')$ we denote the set of tasks that are simultaneously successors of t and predecessors of t', while $I^x(t,t')$ denotes $I(t,t') - \{x\}$. For instance, in the task system of Figure 5.1, $I(t_{13},t_4)$ consists of the tasks t_{12}, t_{11}, t_6, t_7, and t_8, while $I^{t_9}(t_{12},t_5) = \{t_{11}$, $t_8\}$. Note that if t and t' are incompatible with respect to $\prec$, then $I(t,t') = \emptyset$. The *scheduling distance* $D(t,t')$ is the length of an optimal two processor schedule for $(I(t,t'), \prec)$. Likewise, $D^x(t,t')$ denotes the length of an optimal two processor schedule for $(I^x(t,t'),\prec)$. Clearly, if t and t' are incompatible with respect to $\prec$, then $D(t,t') = 0$. Let $D(t,A)$ denote $\min\{D(t,t')\,|\,t' \in A\}$, and let $D(A,t')$ denote $\min\{D(t,t')\,|\,t \in A\}$. Let $D^x(t,A)$ and $D^x(A,t)$ be defined equivalently.

The *level of x relative to t'*, denoted by $level_{t'}(x)$, is the length of a longest path from x to t'. Let $U(t,t',\ell)$ be the set of tasks in $I(t,t')$ that are on a level higher than ℓ relative to t'. Note that for any task $x \in I(t,t')$, the level of x (relative to t') is between 1 and $level_{t'}(t) - 1$, and the level of t' (relative to t') is 0. It furthermore holds that $U(t,t',0) = I(t,t')$ and $U(t,t',level_{t'}(t) - 1) = \emptyset$. Whenever t' is clear from the context, we say "level of x" and mean $level_{t'}(x)$. A task is called *critical* in $I(t,t')$ if it is contained in a longest path in $(I(t,t'),\prec$). The set of all critical tasks in $I(t,t')$ on level ℓ is denoted by $Crit(t,t',\ell)$. For instance, in Figure 5.1, tasks t_{12}, t_{11}, and t_8 are the only critical tasks in $I(t_{13},t_4)$, and they are also the only critical tasks in $I(t_{14},t_5)$. The level of t_{11} relative to t_4 is 2, while $U(t_{12},t_1,1) = \{t_6, t_7, t_8, t_{11}\}$ and $Crit(t_{14},t_5,1) = \{t_8\}$.

We say that x is *maximal* in A if $x \in A$ and x has no successor in A. Conversely, x is *minimal* in A if $x \in A$ and x has no predecessor in A. The set of minimal tasks in A is denoted by $\min(A)$ and the set of maximal tasks by $\max(A)$. Let B be a set of integers. If B contains more than one element, then $\min_2 B$ denotes the second smallest element in B. Otherwise, $\min_2 B$ denotes the one element of B.

5.3 Which Levels Jump?

Let $(T,\prec)$ be a task system consisting of L levels. The first step in computing an LMJ schedule is to determine which levels jump. Clearly, the lexicographically maximum jump sequence for $(T,\prec)$ is unique. Given a jump sequence, we can determine which levels jump, since a level ℓ jumps iff the number of tasks on level ℓ minus the number of occurrences of ℓ in the jump sequence is odd. Hence, if level ℓ jumps to level ℓ' in some LMJ schedule for $(T,\prec)$, then ℓ jumps to ℓ' in all LMJ schedules for $(T,\prec)$.

Let $(T,\prec)^{\geq\ell}$ denote the task system that consists of tasks on level ℓ and all tasks on levels above ℓ in $(T,\prec)$. In the following we consider the relationship between schedules for $(T,\prec)$ and schedules for $(T,\prec)^{\geq\ell}$. A level ℓ' with $\ell' \geq \ell$ in $(T,\prec)$ corresponds to level $\ell' - \ell + 1$ in $(T,\prec)^{\geq\ell}$. For the sake of notational convenience, we write "level ℓ'" and refer either to level ℓ' in $(T,\prec)$ or to its corresponding level $\ell' - \ell + 1$ if we are considering the restricted task system at that moment.

Lemma 5.2 *Let S be an LMJ schedule for $(T,\prec)$ and let S' be the restriction of S to tasks on levels $L,\ldots,\ell$. Then S' is an LMJ schedule for $(T,\prec)^{\geq\ell}$.*

Proof. Those levels in $L,\ldots,\ell+1$ that jump to a task above $\ell-1$ in the LMJ schedule for $(T,\prec)$ can jump to the same task in an LMJ schedule for $(T,\prec)^{\geq\ell}$, and vice versa. Consequently, the same levels above $\ell-1$ jump in an LMJ schedule for $(T,\prec)$ and in an LMJ schedule for $(T,\prec)^{\geq\ell}$. Every level in $L,\ldots,\ell$ that jumps to a task on a level below ℓ in the LMJ schedule for $(T,\prec)$ will jump to level 0 in an LMJ schedule for $(T,\prec)^{\geq\ell}$. It follows that the restriction of an LMJ schedule for $(T,\prec)$ to tasks on levels $L,\ldots,\ell$ is an LMJ schedule for those tasks. □

Levels that jump in LMJ schedules are closely related to *solitary tasks*.

Definition 5.3 *A task t on level ℓ is* solitary *if there is an LMJ schedule for $(T,\prec)^{\geq\ell}$ such that t is the only task mapped to the last timestep.*

For instance, in the task system depicted in Figure 5.1, the solitary tasks are t_{15}, t_{12}, t_{11}, t_9, t_8, t_7, t_6, t_5, and t_4. Task t_{10} is not solitary since it is the only task that can be paired with task t_{15}, and as a consequence, it is used as a fill-in task for the jump from level 7 in every LMJ schedule. Tasks t_{14}, t_{13}, t_3, t_2, and t_1 are not solitary since level 6 and level 1 do not jump.

We show in the following that a level jumps in an LMJ schedule iff there exists at least one solitary task on that level.

Lemma 5.4 *Let x be the task that is used to jump from level ℓ in an LMJ schedule S for $(T,\prec)$. Then x is solitary.*

Proof. Let S' be the restriction of S to tasks on levels $L,\ldots,\ell$. By Lemma 5.2, S' is an LMJ schedule for $(T,\prec)^{\geq\ell}$. Since ℓ jumps to a level below ℓ in S, level ℓ jumps to level 0 in S'. Hence, x is the only task mapped to the last timestep of S'. As a consequence, x is solitary. □

Lemma 5.5 *Let x be a solitary task on level ℓ. Then ℓ jumps in an LMJ schedule S for $(T,\prec)$.*

Proof. Assume that ℓ does not jump in S. Let S' be the restriction of S to tasks on levels $L,\ldots,\ell$. Clearly, S' is an LMJ schedule for $(T,\prec)^{\geq\ell}$ (Lemma 5.2). Since level ℓ does not jump in S, it does not jump in S'. It follows that ℓ does not jump in any LMJ schedule for $(T,\prec)^{\geq\ell}$. Hence, no solitary task exists on level ℓ. A contradiction. We conclude that our assumption is wrong and ℓ jumps in an LMJ schedule for $(T,\prec)$. □

To decide which tasks are solitary, we use the following construction. Let G be a precedence graph for $(T,\prec)$. We augment G as follows (cf. Figure 5.3). Let α be a new task that precedes all other tasks. For $\ell = 2,\ldots,L$, let γ_ℓ and $\hat{\gamma}_\ell$ be two new tasks that are successors of all tasks in T on level ℓ and above. Let γ_{L+1} and $\hat{\gamma}_{L+1}$ be two new tasks that are successors of α. Let γ_1 be a new task that is successor of all tasks in T. For every task $x \in T$, let β_x be a new task that is successor of $\gamma_{level(x)+1}$ and $\hat{\gamma}_{level(x)+1}$ and successor of all tasks in T on the level of x but excluding x. Let $(T',\prec')$ denote the task system obtained by this construction. The solitary tasks of the original task system $(T,\prec)$ can now be characterized as follows.

Lemma 5.6 *A task x on level ℓ in $(T,\prec)$ is solitary iff $D(\alpha,\beta_x) = D(\alpha,\gamma_\ell)$.*

Proof. Let x be a solitary task on level ℓ. Then there exists an LMJ schedule S for $(I(\alpha,\gamma_\ell),\prec')$ such that x is the only task mapped to the last timestep.

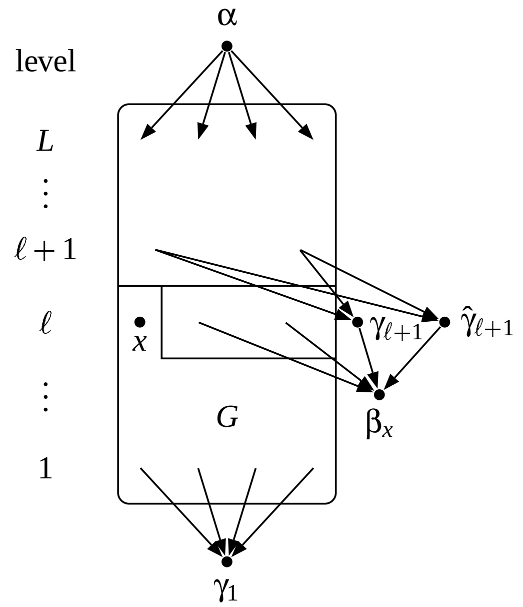


Figure 5.3: *To decide which levels jump, we augment G by* $2L+n+2$ *tasks:* γ_1 *succeeds all tasks of* T*,* γ_ℓ *and* $\hat{\gamma}_\ell$ *succeed all tasks on levels* $\geq \ell$ *(for* $\ell = 2,\ldots,L+1$*),* β_x *succeeds* $\gamma_{level(x)+1}$ *and* $\hat{\gamma}_{level(x)+1}$ *and all tasks on the level of x with the exception of x (for every* $x \in T$*), and* α *precedes all tasks.*

Instead of x we schedule $\gamma_{\ell+1}$ and $\hat{\gamma}_{\ell+1}$ at timestep $S(x)$ to obtain a schedule S' for $(I(\alpha,\beta_x),\prec')$. Since $\gamma_{\ell+1}$ and $\hat{\gamma}_{\ell+1}$ can not be used as fill-ins, S' is an LMJ schedule and hence optimal. Clearly, S and S' have equal length, therefore $D(\alpha,\beta_x) = D(\alpha,\gamma_\ell)$. To prove the reverse implication, let $D(\alpha,\beta_x) = D(\alpha,\gamma_\ell)$ and assume that x is not solitary. We have to consider two cases.

case 1: There exists an LMJ schedule S for $(I(\alpha,\gamma_\ell),\prec')$ such that x is not used as a fill-in. Then level ℓ does not jump, since otherwise we could easily modify S such that x is the only task mapped to the last timestep, in which case x would be solitary. We replace task x by $\gamma_{\ell+1}$ and map task $\hat{\gamma}_{\ell+1}$ to timestep $length(S)+1$ to obtain a schedule S' for $(I(\alpha,\beta_x),\prec')$. Since $\gamma_{\ell+1}$ and $\hat{\gamma}_{\ell+1}$ can not be used as fill-ins, S' is an LMJ schedule and hence optimal. Because S' is one timestep longer than S, we obtain $D(\alpha,\beta_x) = D(\alpha,\gamma_\ell)+1$, a contradiction. Hence, our assumption is wrong and x is solitary in this case.

case 2: x is used as a fill-in in every LMJ schedule for $(I(\alpha,\gamma_\ell),\prec')$. In other words, among the levels $\ell_1,\ldots,\ell_k$ that jump to level ℓ in an LMJ schedule for $(I(\alpha,\gamma_\ell),\prec')$, only $k-1$ will find a suitable fill-in on level ℓ if we remove x. Furthermore note that none of the levels $\ell_1,\dots,\ell_k$ can use $\gamma_{\ell+1}$ or $\hat{\gamma}_{\ell+1}$ as fill-in. Hence, in an LMJ schedule for $(I(\alpha,\beta_x),\prec')$, one of $\ell_1,\dots,\ell_k$ will jump to level 0 and only $k-1$ tasks from level ℓ will be used as fill-ins. As a consequence, the number of tasks on level ℓ not used as fill-ins increases by two if we remove x and add $\gamma_{\ell+1}$ and $\hat{\gamma}_{\ell+1}$. Clearly, a schedule for $(I(\alpha,\gamma_\ell),\prec')$ consists of $D(\alpha,\gamma_{\ell+1})$ timesteps required to schedule tasks on levels $L,\dots,\ell+1$ plus $\lceil \frac{i}{2} \rceil$ timesteps for the i tasks on level ℓ not used as fill-ins. In an LMJ schedule for $(I(\alpha,\beta_x),\prec')$ the number of tasks from level ℓ not used as fill-ins is $i+2$, as we have just observed, while the number of timesteps required to schedule the tasks on levels $L,\dots,\ell+1$ is still $D(\alpha,\gamma_{\ell+1})$. We obtain $D(\alpha,\beta_x) = D(\alpha,\gamma_\ell)+1$. A contradiction. We conclude that our assumption is wrong and x is solitary. □

In the following three sections we are concerned with the problem of determining the scheduling distances $D(t,t')$ for every pair of tasks. In the next section and the section thereafter we establish properties of task systems that our distance algorithm depends on.

5.4 The Scheduling Distance

We start with a basic observation on the length of optimal two processor schedules. Throughout this section and the next two sections let t and t' be two arbitrary tasks of some task system $(T,\prec)$.

Lemma 5.7 *Let $A \subseteq I(t,t')$. Then*

$$D(t,t') \geq D(t,A) + D(A,t') + \left\lceil \frac{|A|}{2} \right\rceil .$$

Proof. In every schedule for $(I(t,t'),\prec)$, at least $D(t,A)$ timesteps are between t and any task of A. Moreover, at least $D(A,t')$ timesteps are between any task of A and t'. Since we require at least $\lceil |A|/2 \rceil$ timesteps to schedule all tasks of A, any schedule for $(I(t,t'),\prec)$ has length at least $D(t,A)+D(A,t')+\lceil |A|/2 \rceil$. □

Note that this lemma still holds if we replace I by I^x and D by D^x, for some task x. An important property of level schedules concerns critical tasks: they are never used as fill-ins for nontrivial jumps.

Lemma 5.8 *No critical task in $I(t,t')$ is a fill-in for a nontrivial jump in any level schedule for $(I(t,t'), \prec)$.*

Proof. Assume there exist critical tasks that are used as fill-ins for nontrivial jumps. Let x be one of the highest such tasks, say x is on level ℓ and level $\ell' > \ell+1$ jumps to ℓ using x as a fill-in. Since x is critical there must be a predecessor y of x on level $\ell+1$ that is critical too. Task y is not a fill-in for a nontrivial jump because we selected x to be one of the highest critical tasks that are fill-ins for nontrivial jumps. It follows that y either is the fill-in for the jump from level $\ell+2$ or is scheduled later than all tasks on level $\ell+2$. Since x is a fill-in for level $\ell' > \ell+1$ we obtain that x is scheduled earlier than y, which violates the precedence constraints. We conclude that no such task x exists. □

In our scheduling distance computation, scheduling distances are computed iteratively from smaller scheduling distances that are already known. An important question in this context is, whether we can find two tasks x and y in $I(t,t')$ such that the sum of $D(t,y)$ and $D(x,t')$ equals $D(t,t')$. Unfortunately, this is not always possible. Clearly, if we choose x and y arbitrarily, we can do arbitrarily bad. A good choice for x and y seems to be one where we know that x and y must be scheduled close to each other, for instance, two critical tasks on successive levels. As it turns out, there actually exist critical tasks x and y such that x is one level higher than y and $D(t,y)+D(x,t')$ equals either $D(t,t')$ or $D(t,t')+1$. The existence of these two particular critical tasks is proved in the next section. In this section, we are concerned about what happens if we choose the wrong two critical tasks on successive levels. The following lemma shows that in this case the above sum exceeds $D(t,t')$ by at most one. Moreover, we give a condition necessary for $D(t,y)+D(x,t')$ to exceed $D(t,t')$.

Lemma 5.9 *Let x and y be critical tasks in $I(t,t')$ such that x is one level higher than y. Then*

1. $D(t,y)+D(x,t') \leq D(t,t')+1$, *and*
2. *if $D(t,y)+D(x,t') = D(t,t')+1$, then there exists an immediate successor z of x in $I(x,t')$ such that $D^z(x,t') < D(x,t')$ and $D(t,z) < D(t,y)$.*

Proof. Consider an optimal level schedule S, and let ℓ denote the level of y (relative to t'). Let τ be the last timestep a task on level $\ell+1$ is mapped to. (1.) According to Lemma 5.8, each of x and y either is not used in a jump or is used as a fill-in for a trivial jump from level $\ell+2$ respectively

$\ell+1$. Therefore, we only have to consider four cases: (a) If level $\ell+1$ does not jump, then $D(t,y) \leq \tau$ and $D(x,t') \leq D(t,t') - \tau$ (Figure 5.4a). Hence, $D(t,y) + D(x,t') \leq D(t,t')$. (b) The same holds if $\ell+1$ jumps and uses x to jump from $\ell+1$ (Figure 5.4b). (c) If $\ell+1$ jumps and uses y as a fill-in, then $D(t,y) \leq \tau - 1$ and $D(x,t') \leq D(t,t') - \tau + 1$ (Figure 5.4c). Again it holds that $D(t,y) + D(x,t') \leq D(t,t')$. (d) Otherwise, $\ell+1$ jumps and uses a task different from x to jump from $\ell+1$ and a task z different from y as fill-in (Figure 5.4d). In this case $D(t,y) \leq \tau$ and $D(x,t') \leq D(t,t') - \tau + 1$. It follows that $D(t,y) + D(x,t') \leq D(t,t') + 1$.

(2.) Let $D(t,y) + D(x,t') = D(t,t') + 1$. Then level $\ell+1$ jumps and a task different from x is used to jump from level $\ell+1$ and a task z different from y is used as fill-in (Figure 5.4d). Clearly, z is a successor of x, $D(t,y) = \tau$, and $D(x,t') = D(t,t') - \tau + 1$. Only tasks from level $\ell+1$ are scheduled between x and z. Hence, z is an immediate successor of x and $D(t,z) + D(x,t') \leq D(t,t')$. It follows that $D(t,z) < D(t,y)$. The task paired with z is from level $\ell+1$ and therefore independent of x. Hence, if we remove z, then the scheduling distance between x and t' decreases by one, *i.e.*, $D^z(x,t') = D(x,t') - 1$. □

Consider tasks t_6 and t_7 in the precedence graph depicted in Figure 5.5. Both tasks are critical in $I(t_{12},t_1)$ and are on successive levels. If we add $D(t_{12},t_6)$ and $D(t_7,t_1)$, then we obtain 6, which exceeds the scheduling distance between task t_{12} and task t_1 by 1. Lemma 5.9.1 guarantees that this is the worst that can happen. To finish the example, observe that task t_5 is a task that can be used to detect this overflow according to Lemma 5.9.2, since t_5 is an immediate successor of t_7, $D(t_{12},t_5) < D(t_{12},t_6)$, and $D^{t_5}(t_7,t_1) < D(t_7,t_1)$.

To recognize that the sum of $D(t,y)$ and $D(x,t')$ exceeds $D(t,t')$, we have to check for each immediate successor z of x in $I(x,t')$ whether removing z decreases $D(x,t')$ and whether the distance between t and z is smaller than the distance between t and y. As it turns out, checking whether removing z decreases $D(x,t')$ for each immediate successor z of x is expensive. For practical purposes, we need a condition that can be checked more easily. The concept that provides such a condition is an "overflow indicator".

Definition 5.10 *Let $F \subseteq I(t,t')$. Then F is an* overflow indicator *for (t,t') if*

$$D(t,t') = \left\lceil \frac{|F|}{2} \right\rceil + D(F,t').$$

Note that in every optimal schedule for $(I(t,t'),\prec)$ the tasks of an overflow indicator F are scheduled in the first $\lceil |F|/2 \rceil$ timesteps. Moreover, if the size

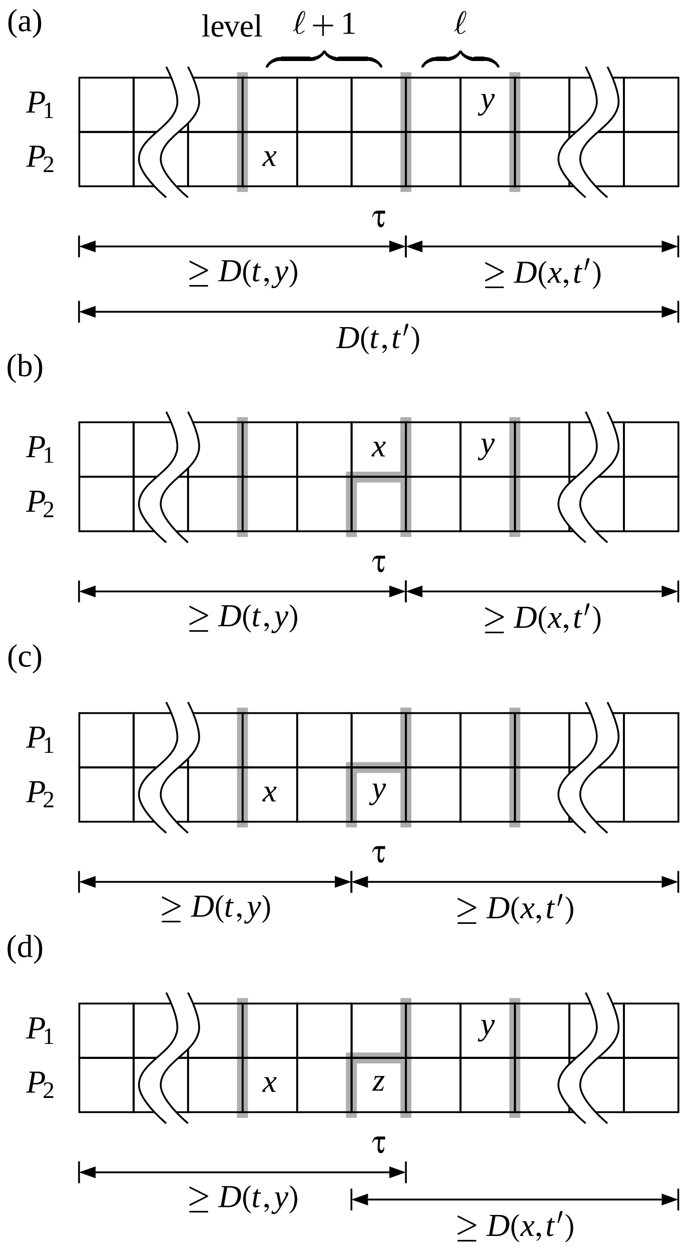
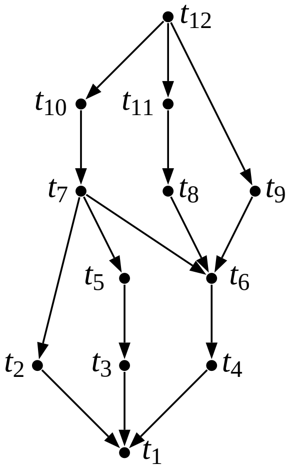


Figure 5.4: *If x and y are critical tasks on successive levels $\ell+1$ and ℓ, then in any optimal level schedule the sum of $D(t,y)$ and $D(x,t')$ exceeds $D(t,t')$ by at most one. There are four cases: (a) level $\ell+1$ does not jump, (b) x is used to jump from level $\ell+1$, (c) y is used as fill-in for the jump from level $\ell+1$, or (d) some other task z is used as fill-in.*



P_1	t_{12}	t_{11}	t_9	t_8	t_6	t_4	t_1
P_2		t_{10}	t_7	t_5	t_2	t_3	
timestep	1	2	3	4	5	6	7

Figure 5.5: *A task system and an optimal level schedule for it. Tasks t_6 and t_7 are critical tasks on successive levels but the sum of $D(t_{12},t_6)$ and $D(t_7,t_1)$ is greater than $D(t_{12},t_1)$.*

of F is even, then two tasks from F are mapped to timestep $|F|/2$ and both of them have the same scheduling distance to t', which is the minimum distance to t' among tasks in F. In the following we show how overflow indicators for (x,t') can be used to detect that $D(t,y)+D(x,t')$ exceeds $D(t,t')$.

Lemma 5.11 *Let x and y be critical tasks in $I(t,t')$ such that x is one level higher than y. Let F be an overflow indicator for (x,t'), and let $D(t,y)+D(x,t') = D(t,t')+1$. Then the size of F is odd and there exists a task $z \in F$ such that $D(t,z) < D(t,y)$.*

Proof. According to Lemma 5.9.2 there exists an immediate successor z of x in $I(x,t')$ such that $D^z(x,t') < D(x,t')$ and $D(t,z) < D(t,y)$. Applying Lemma 5.7 to $F-\{z\}$ and $I^z(x,t')$, we obtain $D^z(x,t') \geq \lceil |F-\{z\}|/2 \rceil + D^z(F-\{z\},t')$. Since z is an immediate successor of x, z is not contained in $I(u,t')$ for any $u \in F$. Therefore, $D^z(u,t') = D(u,t')$. We obtain

$$D^z(x,t') \geq \left\lceil \frac{|F-\{z\}|}{2} \right\rceil + D(F-\{z\},t'). \tag{5.1}$$

Assume that z is not contained in F. Then the right side of (5.1) becomes $\lceil |F|/2 \rceil + D(F,t')$, and that equals $D(x,t')$ since F is an overflow indicator for (x,t'). As a consequence, $D^z(x,t') \geq D(x,t')$, which contradicts $D^z(x,t') <$

$D(x,t')$. Hence, our assumption is wrong and we conclude that z is contained in F.

Assume next that the size of F is even. Since F is an overflow indicator for (x,t') we know that $D(x,t') = \lceil |F|/2 \rceil + D(F,t')$. As noted before, in an optimal schedule for $I(x,t')$, the tasks of F are scheduled in the first $|F|/2$ timesteps. Since $|F|$ is even there are at least two tasks in F with minimum scheduling distance to t'. Hence, removing one task from F does not change the minimum distance to t'. It is furthermore clear that $\lceil |F|/2 \rceil = \lceil |F - \{z\}|/2 \rceil$. Applying these facts to (5.1), we obtain $D^z(x,t') \geq \lceil |F|/2 \rceil + D(F,t')$, and that equals $D(x,t')$. Again we derive $D^z(x,t') \geq D(x,t')$, contradicting the fact that $D^z(x,t') < D(x,t')$. Hence, our assumption is wrong, showing that the size of F is odd. □

What remains to consider is whether overflow indicators can mislead us. If the size of some overflow indicator F for (x,t') is odd and F contains a task z such that $D(t,z) < D(t,y)$, does that imply that $D(t,y) + D(x,t')$ exceeds $D(t,t')$? The answer is *no*, but one can show that there exist critical tasks x and y on successive levels such that $D(t,y) + D(x,t')$ equals $D(t,t')$ and there exists at least one overflow indicator F for (x,t') such that either the size of F is even or no task in F has a shorter distance to t than y. Moreover, such an overflow indicator can be found efficiently. We will deal with this issue in section 5.6.

5.5 Block Decompositions

In this section we take a closer look at the sequential two processor scheduling algorithm of Coffman and Graham [CG72]. The analysis of this algorithm provides us with necessary means to compute scheduling distances in parallel. In their algorithm, tasks are labeled according to a lexicographic numbering scheme. Then, tasks are put into a list in decreasing order of these numbers, and finally a list schedule for this list is computed. In the following we give a short review of the algorithm and its correctness proof.

Each task x will be given a distinct number *label*(x) in the range $1,\ldots,n$. If all immediate successors of x already have a label, then define *ll*(x) to be the list of these numbers in decreasing order. We call *ll*(x) the *label list* of x. If x has no successors, then *ll*(x) is the empty list. In what follows, the label lists of tasks are compared lexicographically, *e.g.*, $(7,6,4,1) < (7,6,5)$ and $(3,1) < (4,3,1)$. In particular, the empty list is smaller than all nonempty lists. To label tasks, repeat the following:

> Consider all tasks x that already have a label list but no label (initially, this is only the case for tasks that have no successors). Among these tasks choose a task x with a lexicographically minimal label list and assign the smallest positive integer to $label(x)$ that has not been assigned to some other task label (hence, the first task obtains number 1).

For instance, a possible labeling for the tasks in Figure 5.1 is: $label(t_1) = 1$, $label(t_2) = 2$, $label(t_3) = 3$, $label(t_4) = 4$, $label(t_5) = 5$, $label(t_6) = 6$, $label(t_9) = 7$, $label(t_{10}) = 8$, $label(t_7) = 9$, $label(t_8) = 10$, $label(t_{11}) = 11$, $label(t_{12}) = 12$, $label(t_{13}) = 13$, $label(t_{14}) = 14$, and $label(t_{15}) = 15$.

Let L be a list of all tasks sorted in decreasing order of labels, and let S be the list schedule for L. Note that S is a level schedule because tasks are labeled in level order, *i.e.*, if x is on a higher level than y, then $label(x) > label(y)$. To prove that S is optimal, we show that S can be split into blocks that have to be processed sequentially in any schedule. We assume that all empty tasks that occur in S have label 0. Let τ_1 be the latest timestep of S, and let v_1 be an arbitrary nonempty task mapped to τ_1. Let w_1 be the other (possibly empty) task mapped to τ_1. We define inductively, as long as $\tau_{i-1} > 0$:

$\tau_i :=$ the latest timestep τ before τ_{i-1} such that there is a task x mapped to τ with $label(x) < label(v_{i-1})$ (if no such timestep exists, then $\tau_i := 0$).

$w_i :=$ the (possibly empty) task mapped to τ_i with the smaller label.

$v_i :=$ the other (nonempty) task mapped to τ_i.

$\chi_{i-1} :=$ the set of tasks mapped to timesteps strictly between τ_i and τ_{i-1} plus v_{i-1}.

Let $\chi_k, \ldots, \chi_1$ be the blocks defined by this procedure, *i.e.*, $\tau_{k+1} = 0$. Note that no timestep strictly between τ_i and τ_{i-1} is partial. Hence, the size of each χ_i is odd. Moreover,

$$length(S) = \sum_{i=1}^{k} \left\lceil \frac{|\chi_i|}{2} \right\rceil .$$

In Figure 5.6, a schedule and the blocks constructed by the above procedure are sketched. Tasks are depicted as dotted squares and each block is framed grey.

To prove that S is of minimal length, the following auxiliary propositions are required.

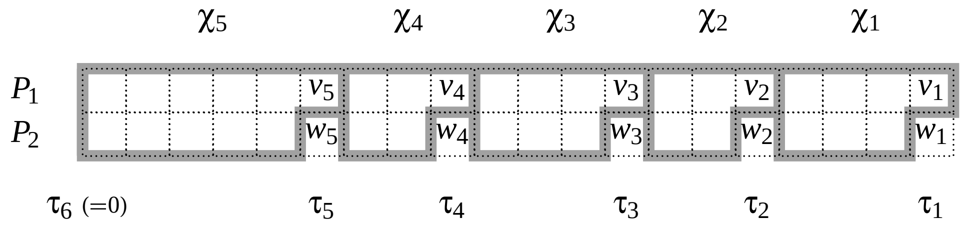


Figure 5.6: *Blocks are constructed from right to left by inductively defining appropriate timesteps τ_i and tasks v_i. Tasks w_i are not contained in blocks and may be empty.*

Lemma 5.12 *The labels of tasks in χ_i are greater than the labels of tasks in $\chi_{i-1}, \ldots, \chi_1$, and greater than the labels of $w_{i+1}, \ldots, w_1$.*

Proof. We first prove that all labels of tasks in χ_i are greater than the labels of tasks in χ_{i-1}. Let $x \in \chi_i$ and let $y \in \chi_{i-1}$. By construction, $label(y) \geq label(v_{i-1})$, $label(x) \geq label(v_i)$, and $label(w_i) < label(v_{i-1})$. Since L contains tasks sorted by decreasing label and v_i is scheduled earlier than y and $label(y) > label(w_i)$, we derive that $label(v_i) > label(y)$. (If this would not be the case, then there would exist a task with greater label than v_i and w_i that is available at timestep τ_i. Hence, the list scheduling algorithm would map this task to τ_i before it considers v_i and w_i.) As a consequence, $label(x) > label(y)$, which proves the claim. Since the claim holds for every $2 \leq i \leq k$, we immediately obtain that the labels of tasks in χ_i are also greater than the labels of all tasks in $\chi_{i-2}, \ldots, \chi_1$. Since $label(w_j) < label(v_{j-1})$, for $2 \leq j \leq k$, we furthermore obtain that the labels in χ_i are greater than the labels of $w_{i+1}, \ldots, w_1$. □

Lemma 5.13 *Every task of χ_i is a predecessor of all tasks in χ_{i-1}, for $i = 2, \ldots, k$.*

Proof. Let $y \in \chi_{i-1}$. As already observed, $label(y) > label(w_i)$. It follows that v_i is a predecessor of y, since otherwise v_i would have been paired with y instead of w_i. As a consequence, v_i is predecessor of all tasks in χ_{i-1}. Let x be a maximal task in χ_i. By Lemma 5.12, the labels of tasks in χ_{i-1} are greater than the labels of tasks in $\chi_{i-2}, \ldots, \chi_1$ and greater than the labels of $w_i, \ldots, w_1$. Therefore, no successor of x has a label that is greater than any label in χ_{i-1}. By construction, $label(x) \geq label(v_i)$. Hence, $ll(x)$ is lexicographically not smaller than $ll(v_i)$. Since v_i is predecessor of all tasks in χ_{i-1}, x must also be predecessor of all tasks in χ_{i-1}. We have shown that the lemma holds for all maximal tasks of χ_i. All other tasks of χ_i have a successor in χ_i, and hence, by transitivity of $\prec$, the lemma holds for them as well. □

Clearly, this lemma implies that S is of minimal length: any schedule requires at least $\sum_{i=1}^{k} \lceil |\chi_i|/2 \rceil$ timesteps to schedule the sets $\chi_k, \ldots, \chi_1$. What is important to our work is the existence of the sets $\chi_k, \ldots, \chi_1$.

Definition 5.14 *Let $(T, \prec)$ be a task system, let S be a schedule for $(T, \prec)$, and let $\chi_k, \ldots, \chi_1$ be pairwise disjoint subsets of T (called* blocks*) such that*

1. *every task of χ_i is a predecessor of all tasks in χ_{i-1},*
2. *the length of S equals $\sum_{i=1}^{k} \left\lceil \frac{|\chi_i|}{2} \right\rceil$,*
3. *the size of each block is odd, and*
4. *the latest task of χ_i in S is not paired with a task of χ_i.*

Then $\chi_k, \ldots, \chi_1$ is called a block decomposition for $(T, \prec)$ with schedule S. *Moreover, v_i denotes the latest task of χ_i in S and w_i denotes the (possibly empty) task that is paired with v_i in S.*

From the discussion above, we immediately obtain

Theorem 5.15 ([CG72]) *For every task system $(T, \prec)$ there exists a block decomposition χ_k, …, χ_1 with schedule S such that*

1. *S is a level schedule and*
2. *the level of w_i is not higher than the level of v_{i-1}, for $2 \leq i \leq k$.*

A block decomposition with a schedule for the precedence graph in Figure 5.1 is given in Figure 5.1. In this example, the decomposition consists of three blocks χ_3, χ_2, χ_1 where $v_1 = t_3$, $w_1 = t_2$, $v_2 = t_{12}$, $v_3 = t_{15}$, $w_3 = t_{10}$, and w_2 is an empty task. Since the schedule is a level schedule, the decomposition conforms to Theorem 5.15. Note, however, that the schedule is not obtained by the algorithm of Coffman and Graham.

Let us continue with our considerations of critical tasks on successive levels that we started in the last section. It is apparent now, how the critical tasks x and y should be chosen such that $D(t,y) + D(x,t') \geq D(t,t')$: they should be contained in successive blocks of some block decomposition for $I(t,t')$.

Lemma 5.16 *Let $\chi_k,\dots,\chi_1$ be a block decomposition for $(I(t,t'),\prec)$. Let $x \in \chi_{j+1}$ and $y \in \chi_j$. Then*

$$D(t,y)+D(x,t') \geq D(t,t').$$

Proof. Since y is contained in χ_j, it is successor of all tasks in $\chi_k,\dots,\chi_{j+1}$. Hence, we require at least $\sum_{i=j+1}^{k} \lceil |\chi_i|/2 \rceil$ timesteps to schedule $I(t,y)$. Conversely, x is predecessor of all tasks in $\chi_j,\dots,\chi_1$, and we require at least $\sum_{i=1}^{j} \lceil |\chi_i|/2 \rceil$ timesteps to schedule $I(x,t')$. Therefore

$$D(t,y)+D(x,t') \geq \sum_{i=j+1}^{k} \left\lceil \frac{|\chi_i|}{2} \right\rceil + \sum_{i=1}^{j} \left\lceil \frac{|\chi_i|}{2} \right\rceil = \sum_{i=1}^{k} \left\lceil \frac{|\chi_i|}{2} \right\rceil = D(t,t').$$

□

We combine this result with Lemma 5.9.1 and obtain that the value of $D(t,y)+D(x,t')$ is either $D(t,t')$ or $D(t,t')+1$ if x and y are critical tasks on successive levels and contained in successive blocks of some block decomposition.

This leaves us with the problem of finding such tasks. Clearly, we have to make sure that the two critical tasks we want to use are actually contained in blocks of some block decomposition. Since we do not want to construct a block decomposition explicitly, we use the following approach. We claim that if a critical task x is not contained in blocks, then the block that contains tasks from the level of x contains only tasks from the level of x. A block that contains only tasks from one level is called *critical*. We will see in Section 5.6 that critical blocks too can be used to determine scheduling distances. As a consequence, we do not depend solely on critical tasks. Either the critical tasks on successive levels are contained in blocks or we can find a critical block. In either way, we are able to determine the scheduling distance.

Lemma 5.17 *Let $\chi_k,\dots,\chi_1$ be a block decomposition for $(T,\prec)$ with schedule S such that S is a level schedule and w_j is not on a higher level than v_{j-1}, for $2 \leq j \leq k$. Moreover, let w_i be a critical task, for some $i > 1$. Then χ_{i-1} is a critical block with tasks from the level of w_i.*

Proof. All tasks outside blocks (except w_1) are fill-ins for jumps, since S is a level schedule, every w_j is not an a higher level than v_{j-1}, and v_{j-1} is a successor of v_j. The only exception is w_1 that is either an empty task or on the same level as v_1. Hence, if a critical task is w_i, for some $i > 1$, then this critical task is a fill-in. Since S is a level schedule and critical tasks can only be used as fill-ins for trivial jumps in level schedules (Lemma 5.8), the highest tasks in χ_{i-1} are on the same level as w_i. Since w_i is not on a higher level than v_{i-1}, which is on the lowest level in χ_{i-1}, we can derive that χ_{i-1} contains only tasks from the level of w_i. As a consequence, χ_{i-1} is a critical block. □

Another question that turns out to be important in our scheduling distance computation is the following. Let $\chi_k,\dots,\chi_1$ be a block decomposition for $(I(t,t'),\prec)$, and let x be a maximal task in χ_i. We wish to determine the number of blocks in a block decomposition for $(I(x,t'),\prec)$. It is not difficult to verify that if $w_i \notin I(x,t')$, then $\chi_{i-1},\dots,\chi_1$ is a block decomposition for $(I(x,t'),\prec)$. But what happens if $w_i \in I(x,t')$? Note that in the symmetric case no problems occur: if y is a minimal task in χ_i, then $\chi_k,\dots,\chi_{i+1}$ is a block decomposition for $(I(t,y),\prec)$.

To handle this problem and to simplify forthcoming proofs in other respects, we modify the schedules and block decompositions of Coffman and Graham. This modification consists of two steps. First, we show that there exists a block decomposition such that w_1 or w_2 is an empty task. Second, we move the tasks $w_k,\dots,w_3$ ($w_k,\dots,w_2$) one block to the right. This is possible since w_2 (w_1) is an empty task and w_i has no successors in χ_{i-1}, for $2 \le i \le k$. The decomposition obtained is called a "canonical block decomposition".

Definition 5.18 *Let $\chi_k,\dots,\chi_1$ be a block decomposition for $(T,\prec)$ with schedule S. Then $\chi_k,\dots,\chi_1$ is a* canonical block decomposition for $(T,\prec)$ with schedule S *if*

1. *w_i has no predecessor in χ_i, for $2 \le i \le k$,*
2. *if w_i is a critical task, then χ_i is a critical block and all tasks in χ_i are on the same level as w_i, for $2 \le i \le k$, and*
3. *w_i is not on a higher level than v_i, for $1 \le i \le k$.*

Lemma 5.19 *For every task system $(T,\prec)$ there exists a block decomposition $\chi_k,\dots,\chi_1$ and a schedule S such that $\chi_k,\dots,\chi_1$ is a canonical block decomposition with schedule S.*

Proof. We first show that we can always find a block decomposition such that w_1 or w_2 is an empty task. Let t be a new task that is successor of all tasks in T, and let $\chi'_k,\dots,\chi'_1$ be a block decomposition for $T \cup \{t\}$ that conforms to Theorem 5.15. In particular, the level of w'_i is less than or equal to the level of v'_{i-1}, for $2 \leq i \leq k$. Since $v'_1 = t$ and t is the only task on level one, w'_2 is an empty task (provided that $k > 1$). If χ'_1 consists of t only, then let $\chi_{k-1}, \ldots, \chi_1 := \chi'_k, \ldots, \chi'_2$. Clearly, $\chi_{k-1}, \ldots, \chi_1$ is a block decomposition for $(T, \prec)$ that conforms to Theorem 5.15 and w_1 is an empty task. Otherwise, let $\chi_k, \ldots, \chi_1 := \chi'_k, \ldots, \chi'_2, \chi'_1 - \{t\}$. Now, $\chi_k, \ldots, \chi_1$ is a block decomposition for $(T, \prec)$ that conforms to Theorem 5.15 and w_2 is an empty task.

Let $\chi_k, \ldots, \chi_1$ be a block decomposition for $(T, \prec)$ with schedule S that conforms to Theorem 5.15 such that w_1 or w_2 is an empty task. Such a decomposition exists, as we have just shown. We use the fact that w_1 or w_2 is an empty task and shift the w_i's one block to the right. More precisely, if w_2 is an empty task, then let S' be the mapping that equals S with the exception that S' maps the tasks w_k, w_{k-1}, ..., w_3 to timesteps $S(w_{k-1})$, $S(w_{k-2})$, ..., $S(w_2)$. Otherwise w_1 is an empty task and we define S' to be the mapping that equals S with the exception that S' maps the tasks w_k, w_{k-1}, ..., w_2 to timesteps $S(w_{k-1})$, $S(w_{k-2})$, ..., $S(w_1)$. In either case, S' is a valid schedule since w_i is not on a higher level than v_{i-1} and therefore w_i has no successor in χ_{i-1}, for $2 \leq i \leq k$. Note, however, that S' may not be a level schedule anymore.

Let $w'_k, \ldots, w'_1$ be the tasks paired with $v_k, \ldots, v_1$ in S'. Note that w'_k is an empty task and $w'_i = w_{i+1}$, for $1 \leq i < k$ if w_1 is an empty task, respectively $2 \leq i < k$ if w_2 is an empty task. Since w_{i+1} has no predecessor in χ_i, w'_i has no predecessor in χ_i, for $2 \leq i < k$. If w_{i+1} is critical, then χ_i is a critical block that contains only tasks from the level of w_{i+1}, according to Lemma 5.17. Hence, if w'_i is critical, then χ_i is a critical block that contains only tasks from the level of w'_i, for $2 \leq i < k$. Moreover, w'_i is not on a higher level than v_i, for $1 \leq i \leq k$, because w'_k is empty and w_{i+1} is not on a higher level than v_i, for $2 \leq i \leq k$. We conclude that $\chi_k, \ldots, \chi_1$ is a canonical block decomposition for $(T, \prec)$ with schedule S'. □

In the rest of this section, we introduce some propositions that turn out to be helpful in the scheduling distance computation.

Lemma 5.20 *Let $\chi_k, \ldots, \chi_1$ be a canonical block decomposition for $(T, \prec)$ with schedule S, and let χ_i contain a task on level ℓ. Moreover, let x be a critical task on level ℓ that is not contained in χ_i. Then $x = w_i$.*

Proof. Let y be a task in χ_i on level ℓ. Task x can not be contained in a block other than χ_i because then x would be either predecessor or successor of y. Hence, $x = w_j$ for some index $j \in \{1, \ldots, k\}$. Let $j > 1$. By Definition 5.18.2, χ_j is a critical block with tasks from level ℓ. It follows that $i = j$, since otherwise the tasks in χ_j would be either predecessors or successors of

y, contradicting the fact that y is on level ℓ too. Now, let $j = 1$. In this case, x is on level 1. Hence, $\ell = 1$. Clearly, χ_1 contains at least one task z from level 1. It follows that $i = j$, since otherwise y would be a predecessor of z, which contradicts the fact that y is on level 1. □

The reason why blocks with tasks from only one level are called critical is that all tasks in a critical block are critical.

Lemma 5.21 *Let $\chi_k, \ldots, \chi_1$ be a canonical block decomposition for $(T, \prec)$ with schedule S, and let χ_i be a critical block. Then all tasks in χ_i are critical.*

Proof. By induction on the block index i. All tasks on the highest level in χ_k are critical. Hence, the lemma holds for $i = k$. Let the lemma hold for all χ_j with $j > i$.

Assume there is a task x in χ_i on level ℓ that is not critical. Let y be a critical task on level $\ell + 1$. Clearly, x has no predecessor on level $\ell + 1$ that is critical since otherwise x would be critical too. Task x is successor of all tasks in the blocks $\chi_k, \ldots, \chi_{i+1}$ and therefore none of the critical tasks on level $\ell + 1$ is contained in $\chi_k, \ldots, \chi_{i+1}$. It is furthermore clear that none of the tasks on level $\ell + 1$ is contained in $\chi_i, \ldots, \chi_1$. Hence, $y = w_j$ for some index $j \in \{2, \ldots, k\}$ (the case $j = 1$ is ruled out since w_1 is on level 1 and y is on a level > 1). By Definition 5.18.2, all tasks in χ_j are on the same level as w_j, which is level $\ell + 1$. It follows that $j > i$, since $\chi_i, \ldots, \chi_1$ only contain tasks from levels 1 to ℓ. By inductive hypothesis, all tasks in χ_j are critical. Since $j > i$ and x is successor of tasks in $\chi_k, \ldots, \chi_{i+1}$, all tasks in χ_j are predecessors of x. As a consequence, x has a critical predecessor on level $\ell + 1$ and therefore x is critical. A contradiction. We conclude that our assumption is wrong and every task in χ_i is a critical task. □

An important property of canonical block decompositions is that they can be "split" in the following sense.

Lemma 5.22 *Let $\chi_k, \ldots, \chi_1$ be a canonical block decomposition for the task system $(I(t,t'), \prec)$ with schedule S. Let x be a maximal task in χ_{i+1}, and let y be a minimal task in χ_i. Then there exists a canonical block decomposition for $(I(t,y), \prec)$ consisting of $k - i$ blocks and one for $(I(x,t'), \prec)$ consisting of i blocks.*

Proof. Clearly, $\chi_k, \ldots, \chi_{i+1}$ is a canonical block decomposition for $(I(t,y), \prec)$ with schedule S restricted to $I(t,y)$, consisting of $k - i$ blocks. Define the mapping $S' : z \mapsto S(z) - S(v_{i+1})$. One can easily verify that $\chi_i, \ldots, \chi_1$ is a canonical

block decomposition for $(I(x,t'),\prec)$ with schedule S', consisting of i blocks.
□

Note that this lemma does not hold for general block decompositions, since x may be a predecessor of w_{i+1}, in which case $\chi_i,\ldots,\chi_1$ may not be a block decomposition for $(I(x,t'),\prec)$.

5.6 The Distance Algorithm

In Figure 5.7, an algorithm is given that computes the scheduling distances $D(t,t')$ for all pairs of tasks of a given task system. Recall that if A is a subset of $I(t,t')$, then $\max(A)$ denotes the set of maximal tasks in A, while $\min(A)$ is the set of minimal tasks in A. Let $d_r(t,A)$ denote $\min\{d_r(t,t')|t' \in \min(A)\}$, and let $d_r(A,t')$ denote $\min\{d_r(t,t')|t \in \max(A)\}$. An outline of the algorithm is given next.

The algorithm starts by assigning $\lceil |I(t,t')|/2\rceil$ to $d_0(t,t')$, which is a "trivial" lower bound for $D(t,t')$. Then the algorithm iterates $\lceil \log n \rceil$ times to compute $d_1(t,t')$, $d_2(t,t')$, ... which is a nondecreasing sequence of lower bounds for $D(t,t')$. In the end, $d_{\lceil \log n \rceil}(t,t')$ equals $D(t,t')$.

The main loop consists of two parts. In the first part, for every pair of tasks t and t' with $t \prec t'$, a set $H(t,t')$ is determined. It is computed as follows. For each level ℓ between 0 and $level_{t'}(t)-2$, we take $U(t,t',\ell)$, *i.e.*, the tasks of $I(t,t')$ that are on a level higher than ℓ (relative to t'), and determine the maximal tasks in $U(t,t',\ell)$ that have minimal approximated distance to t'. Let M be the set of these tasks. If the size of $U(t,t',\ell)$ is even and M consists of only one task, then let $F(\ell)$ be the set $U(t,t',\ell)-M$, otherwise let $F(\ell)$ equal $U(t,t',\ell)$. Let $H(t,t')$ be the set $F(\ell')$ for the highest level ℓ' such that

$$d_{r-1}(t,t') = \left\lceil \frac{|F(\ell')|}{2} \right\rceil + d_{r-1}(F(\ell'),t').$$

If no level ℓ' between 0 and $level_{t'}(t)-2$ exists such that this equation holds, then let $H(t,t')$ be the empty set.

This set has two important properties. When, at some stage of the iteration, the approximated distance $d_{r-1}(t,t')$ equals $D(t,t')$, then $H(t,t')$ is either empty (in case $d_{r-1}(F(\ell'),t')$ has still the wrong value for all levels ℓ') or it is an overflow indicator for (t,t'). In addition, if $d_{r-1}(x,y)$ has the correct value for all pairs (x,y) where $(I(x,y),\prec)$ has a block decomposition with at most k blocks and there exists a block decomposition for $(I(t,t'),\prec)$ consisting of at most k blocks, then $H(t,t')$ is not only an overflow indicator for (t,t') but it also has a very special form. Namely, either its size is even or its minimal tasks are a subset of the minimal tasks of the first block of a block decomposition for $(I(t,t'),\prec)$. The importance of these properties will become clearer in a moment.

Let us turn to the second part of the main loop. In this part, the new distance approximations $d_r(t,t')$ are computed for all pairs of tasks with $t \prec t'$. This second part consists of three stages. In the first stage, for each level between 1 and $level_{t'}(t)-1$, an approximation $a(\ell)$ based on a critical block is determined. In the second stage, for each level between 1 and $level_{t'}(t)-2$, an approximation $b(\ell)$ based on two critical tasks is determined. In the last stage, the maximum of all approximations is assigned to $d_r(t,t')$.

The approximation $a(\ell)$ based on a critical block is computed as follows. Let p be the second smallest approximated distance between t and a critical task on level ℓ, and let q be the second smallest approximated distance between a critical task on level ℓ and t'. Let C be the set of all critical tasks on level ℓ with a distance to t of at least p and a distance to t' of at least q. Then we compute

$$a(\ell) := d_{r-1}(t,C) + d_{r-1}(C,t') + \left\lceil \frac{|C|}{2} \right\rceil .$$

As we will show, $a(\ell)$ never exceeds $D(t,t')$. On the other hand, if there exists a block decomposition for $(I(t,t'),\prec)$ with a critical block that contains tasks from level ℓ, then, at some stage during the iteration, $a(\ell)$ becomes $D(t,t')$.

The approximation $b(\ell)$ based on two critical tasks is computed as follows. Let $t_1, \ldots, t_{level_{t'}(t)-1}$ be a longest path in $(I(t,t'),\prec)$ such that t_ℓ is on level ℓ relative to t'. Note that all tasks on this path are critical tasks. We compute $b(\ell)$ from $d_{r-1}(t,t_\ell)$ and $d_{r-1}(t_{\ell+1},t')$ as follows. If the set $H(t_{\ell+1},t')$ is empty, or its size is odd and one of its minimal tasks has a shorter approximated distance to t than t_ℓ, then we assign $d_{r-1}(t,t_\ell) + d_{r-1}(t_{\ell+1},t') - 1$ to $b(\ell)$. Otherwise, $b(\ell)$ is $d_{r-1}(t,t_\ell) + d_{r-1}(t_{\ell+1},t')$. We will show that $b(\ell)$ never exceeds $D(t,t')$. On the other hand, if there exists a block decomposition for $(I(t,t'),\prec)$ such that $t_{\ell+1}$ and t_ℓ are contained in different blocks, then, at some stage during the iteration, $b(\ell)$ equals $D(t,t')$.

In what follows we sketch the basic ideas that lead to the correctness proof. For every pair of tasks in the given task system, there exists a canonical block decomposition, according to Lemma 5.19. We do not compute decompositions explicitly but we use their existence to determine $D(t,t')$. Assume that at the beginning of iteration r the values $d_{r-1}(t,t')$ are already the correct dis-

```
ALGORITHM DISTANCE
Input: A task system $(T,\prec)$ with n tasks.
Output: The scheduling distances $D(t,t')$ for all pairs of tasks.
begin
    $d_0(\star,\star) := \left\lceil \frac{|I(\star,\star)|}{2} \right\rceil$;
    for $r := 1$ to $\lceil \log n \rceil$ do
        for all $t, t'$ with $t \prec t'$ do in parallel
            for $\ell := 0$ to $level_{t'}(t) - 2$ do in parallel
                $M := \{x \in \max(U(t,t',\ell)) \,|\, d_{r-1}(x,t') = d_{r-1}(U(t,t',\ell),t')\}$;
                if $|U(t,t',\ell)|$ is even and $|M| = 1$ then
                    $F(\ell) := U(t,t',\ell) - M$;
                else
                    $F(\ell) := U(t,t',\ell)$;
            $H(t,t') := F(\ell')$ for the highest level $\ell'$ such that
                        $d_{r-1}(t,t') = \left\lceil \frac{|F(\ell')|}{2} \right\rceil + d_{r-1}(F(\ell'),t')$
                        or $\emptyset$ if no such level exists;
        for all $t, t'$ with $t \prec t'$ do in parallel
            for $\ell := 1$ to $level_{t'}(t) - 1$ do in parallel
                $p := \min_2\{d_{r-1}(t,x) \,|\, x \in Crit(t,t',\ell)\}$;
                $q := \min_2\{d_{r-1}(x,t') \,|\, x \in Crit(t,t',\ell)\}$;
                $C := \{x \in Crit(t,t',\ell) \,|\, d_{r-1}(t,x) \geq p, d_{r-1}(x,t') \geq q\}$;
                $a(\ell) := d_{r-1}(t,C) + d_{r-1}(C,t') + \left\lceil \frac{|C|}{2} \right\rceil$;
            $t_1,\ldots,t_{level_{t'}(t)-1}$ := a longest path in $(I(t,t'),\prec)$ with $level_{t'}(t_\ell) = \ell$;
            for $\ell := 1$ to $level_{t'}(t) - 2$ do in parallel
                if $H(t_{\ell+1},t') = \emptyset$ or
                    $(d_{r-1}(t,H(t_{\ell+1},t')) < d_{r-1}(t,t_\ell)$ and $|H(t_{\ell+1},t')|$ is odd)
                then
                    $b(\ell) := d_{r-1}(t,t_\ell) + d_{r-1}(t_{\ell+1},t') - 1$;
                else
                    $b(\ell) := d_{r-1}(t,t_\ell) + d_{r-1}(t_{\ell+1},t')$;
            $d_r(t,t') := \max\{d_{r-1}(t,t'),$
                        $\max_{1\leq\ell\leq level_{t'}(t)-1}\{a(\ell)\}, \max_{1\leq\ell\leq level_{t'}(t)-2}\{b(\ell)\}\}$;
    return $d_{\lceil \log n \rceil}(\star,\star)$
end
```

Figure 5.7: *The distance algorithm.*

tances $D(t,t')$ for each set $I(t,t')$ with a canonical block decomposition that consists of at most 2^{r-1} blocks. In iteration r, the following is performed in parallel for each pair of tasks t and t'. Let $\chi_k,\dots,\chi_1$ be a canonical block decomposition for $I(t,t')$ such that $k \leq 2^r$. We "guess" a level ℓ in $I(t,t')$ such that ℓ is the highest level of $\chi_{\lceil k/2 \rceil}$. There are two cases that have to be considered.

If both $t_{\ell+1}$ and t_ℓ are contained in blocks, then t_ℓ is contained in $\chi_{\lceil k/2 \rceil}$ and $t_{\ell+1}$ is contained in $\chi_{\lceil k/2 \rceil+1}$. In this case, $d_{r-1}(t,t_\ell)$ equals $D(t,t_\ell)$ and $d_{r-1}(t_{\ell+1},t')$ equals $D(t_{\ell+1},t')$, since there exist canonical block decompositions for $I(t,t_\ell)$ and $I(t_{\ell+1},t')$ that consist of at most $\left\lceil \frac{k}{2} \right\rceil$ blocks each, and $\left\lceil \frac{k}{2} \right\rceil \leq 2^{r-1}$. Since t_ℓ is minimal in $\chi_{\lceil k/2 \rceil}$ and $t_{\ell+1}$ is maximal in $\chi_{\lceil k/2 \rceil+1}$ and $w_{\lceil k/2 \rceil+1}$ is not a successor of $t_{\ell+1}$, we obtain that $d_{i-1}(t,t_\ell)+d_{r-1}(t_{\ell+1},t')$ equals $D(t,t')$. To distinguish this case from other cases where we "guessed wrong" and ℓ is not the highest level of some block and the sum $d_{i-1}(t,t_\ell)+d_{r-1}(t_{\ell+1},t')$ may exceed $D(t,t')$, we use the set $H(t_{\ell+1},t')$. This overflow indicator has the property that its size is even or no minimal task in it has a shorter approximated distance to t than t_ℓ if ℓ is the highest level of $\chi_{\lceil k/2 \rceil}$. On the other hand, if ℓ is not the highest level of $\chi_{\lceil k/2 \rceil}$ and for some reason $d_{i-1}(t,t_\ell)+d_{r-1}(t_{\ell+1},t')$ exceeds $D(t,t')$, then the size of $H(t_{\ell+1},t')$ is odd and at least one minimal task in it has a shorter approximated distance to t than t_ℓ.

If t_ℓ or $t_{\ell+1}$ is not contained in a block, then $\chi_{\lceil k/2 \rceil}$ or $\chi_{\lceil k/2 \rceil+1}$ is a critical block. Let χ be either $\chi_{\lceil k/2 \rceil}$ or $\chi_{\lceil k/2 \rceil+1}$, whichever of them is critical. For every task x in χ it holds that $d_{r-1}(t,x)$ equals $D(t,x)$ and $d_{r-1}(x,t')$ equals $D(x,t')$ since there exist canonical block decompositions for $I(t,x)$ and $I(x,t')$ that consist of at most $\left\lceil \frac{k}{2} \right\rceil$ blocks each. We compute the minimum distance between t and any task in χ plus the minimum distance of any task in χ to t' plus $\lceil |\chi| / 2 \rceil$, and thus obtain $D(t,t')$.

Since we are unable to guess the "right" level ℓ and to decide whether the chosen critical tasks t_ℓ and $t_{\ell+1}$ are contained in blocks, we perform the above operations for all levels and all supposed critical blocks in parallel. In the end of iteration r, the maximum of all distance approximations obtained this way equals $D(t,t')$, since at least one of the considered levels is the highest level in block $\chi_{\lceil k/2 \rceil}$ and at least one of the considered critical blocks is $\chi_{\lceil k/2 \rceil}$ or $\chi_{\lceil k/2 \rceil+1}$ if one of them is critical.

In the end of iteration r, all values $d_r(t,t')$ equal the correct distances $D(t,t')$ for each set $I(t,t')$ with a block decomposition that consists of at most 2^r blocks. Hence, the number of blocks in block decompositions for which the scheduling distance is known doubles with every iteration. Clearly, $d_0(t,t')$ equals $D(t,t')$ for all sets $I(t,t')$ with a block decomposition consisting of a single block. It follows that after $\lceil \log n \rceil$ iterations, all scheduling distances are known, since no block decomposition consists of more than n blocks.

In the following we prove that the distance algorithm is correct. We first show that the approximated distances never exceed the scheduling distances. Second, we have to show that after $\lceil \log n \rceil$ iterations, the approximated distances equal the scheduling distances.

Lemma 5.23 *For all $r \geq 0$ and all tasks t and t' it holds that $d_r(t,t') \leq D(t,t')$.*

Proof. By induction on r. Clearly, the lemma holds for $r = 0$. Let it hold for some $r-1$, and let ℓ be a level in $I(t,t')$. We first consider the value of $a(\ell)$. We apply Lemma 5.7 to the set C and obtain $D(t,C) + D(C,t') + \lceil |C|/2 \rceil \leq D(t,t')$. Since, by inductive hypothesis, $d_{r-1}(t,t') \leq D(t,t')$ for all pairs of tasks, we obtain that $a(\ell) \leq D(t,t')$.

We now consider the value of $b(\ell)$. Let $t_1, \ldots, t_{level_{t'}(t)-1}$ denote the longest path in $I(t,t')$ chosen by the distance algorithm. We can assume that $d_{r-1}(t,t_\ell)$ + $d_{r-1}(t_{\ell+1},t') > D(t,t')$, since otherwise $b(\ell)$ is clearly a lower bound for $D(t,t')$. By inductive hypothesis, $d_{r-1}(t,t_\ell)$ is a lower bound for $D(t,t_\ell)$ and $d_{r-1}(t_{\ell+1},t')$ is one for $D(t_{\ell+1},t')$. According to Lemma 5.9.1 it holds that the sum $D(t,t_\ell)$ + $D(t_{\ell+1},t')$ exceeds $D(t,t')$ by at most one, hence $d_{r-1}(t,t_\ell)$ + $d_{r-1}(t_{\ell+1},t')$ exceeds $D(t,t')$ by exactly one. It follows that $d_{r-1}(t,t_\ell)$ equals $D(t,t_\ell)$ and $d_{r-1}(t_{\ell+1},t')$ equals $D(t_{\ell+1},t')$. If $H(t_{\ell+1},t')$ is empty, then the algorithm assigns $d_{r-1}(t,t_\ell)$ + $d_{r-1}(t_{\ell+1},t')$ − 1 to $b(\ell)$. In this case, $b(\ell)$ is not greater than $D(t,t')$. Otherwise $H(t_{\ell+1},t')$ equals $F(\ell')$ for some level ℓ' where $d_{r-1}(t_{\ell+1},t') = \lceil |F(\ell')|/2 \rceil$ + $d_{r-1}(F(\ell'),t')$. It holds that $d_{r-1}(F(\ell'),t')$ $\leq D(F(\ell'),t')$ because each $d_{r-1}(y,t')$ is a lower bound for $D(y,t')$, by inductive hypothesis. Hence,

$$D(t_{\ell+1},t') = d_{r-1}(t_{\ell+1},t') \leq \left\lceil \frac{|F(\ell')|}{2} \right\rceil + D(F(\ell'),t').$$

On the other hand, by applying Lemma 5.7 to $F(\ell')$ we obtain

$$D(t_{\ell+1},t') \geq \left\lceil \frac{|F(\ell')|}{2} \right\rceil + D(F(\ell'),t').$$

By combining the last two inequalities, we derive that $D(t_{\ell+1},t') = \lceil |F(\ell')|/2 \rceil$ + $D(F(\ell'),t')$. Hence, $H(t_{\ell+1},t')$ is an overflow indicator for $(t_{\ell+1},t')$ because $H(t_{\ell+1},t') = F(\ell')$. By applying Lemma 5.11, we obtain that $|H(t_{\ell+1},t')|$ is odd and $D(t,H(t_{\ell+1},t')) < D(t,t_\ell)$, a condition recognized by the algorithm because $d_{r-1}(t,H(t_{\ell+1},t'))$ is a lower bound for $D(t,H(t_{\ell+1},t'))$, by inductive hypothesis, and $D(t,t_\ell) = d_{r-1}(t,t_\ell)$, as observed above. As a consequence, the algorithm assigns $D(t,t_\ell) + D(t_{\ell+1},t') - 1$ to $b(\ell)$, which is a lower bound for $D(t,t')$ by Lemma 5.9.1.

The lemma follows from the fact that $d_r(t,t')$ is assigned the maximum of $d_{r-1}(t,t')$, $a(\ell)$, and $b(\ell)$ for all levels ℓ, and all of these approximations are bounded above by $D(t,t')$. □

To prove that the approximated distances eventually converge, we require some auxiliary lemmas. In the first of them we show that the set $H(x,t')$ can be used to decide whether $d_{r-1}(t,y) + d_{r-1}(x,t')$ exceeds $D(t,t')$ if x and y are on successive levels and contained in successive blocks.

Lemma 5.24 *Let $\chi_k,\ldots,\chi_1$ be a canonical block decomposition for the task system $(I(t,t'),\prec)$ with schedule S. For all tasks u and v, let $d_{r-1}(u,v) = D(u,v)$ if there exists a canonical block decomposition for $(I(u,v),\prec)$ consisting of $\leq \lceil \frac{k}{2} \rceil$ blocks. Let x be a maximal task in $\chi_{\lceil k/2\rceil+1}$, and let y be a minimal task in $\chi_{\lceil k/2\rceil}$. Then*

1. *$H(x,t') \neq \emptyset$ and*
2. *the size of $H(x,t')$ is even or $d_{r-1}(t,H(x,t')) \geq d_{r-1}(t,y)$.*

Proof. Let us first note that $d_{r-1}(t,y) = D(t,y)$ and $d_{r-1}(x,t') = D(x,t')$ since, according to Lemma 5.22, there exists a canonical block decomposition for $I(t,y)$ with at most $\lceil \frac{k}{2} \rceil$ blocks and one for $I(x,t')$ with at most $\lceil \frac{k}{2} \rceil$ blocks. Let z be a maximal task in $\chi_{\lceil k/2\rceil}$. By Lemma 5.22, there exists a canonical block decomposition for $I(z,t')$ consisting of at most $\lceil \frac{k}{2} \rceil - 1$ blocks. Hence, $d_{r-1}(z,t') = D(z,t')$. As a consequence,

$$d_{r-1}(\chi_{\lceil k/2\rceil},t') = D(\chi_{\lceil k/2\rceil},t'). \tag{5.2}$$

The same argument can be applied to a minimal task z in $\chi_{\lceil k/2\rceil}$. According to Lemma 5.22 there exists a canonical block decomposition for $I(t,z)$ consisting of at most $\lceil \frac{k}{2} \rceil$ blocks. Therefore $d_{r-1}(t,z) = D(t,z)$. We obtain

$$d_{r-1}(t,\chi_{\lceil k/2\rceil}) = D(t,\chi_{\lceil k/2\rceil}). \tag{5.3}$$

Note that $\chi_{\lceil k/2\rceil}$ is an overflow indicator for (x,t'), *i.e.*, $D(x,t') = \lceil |\chi_{\lceil k/2\rceil}|/2 \rceil + D(\chi_{\lceil k/2\rceil},t')$, since $\chi_{\lceil k/2\rceil}$ is the first block of a block decomposition for $I(x,t')$ (cf. Lemma 5.22). Using (5.2) and the fact that $d_{r-1}(x,t') = D(x,t')$, we obtain

$$d_{r-1}(x,t') = \left\lceil \frac{|\chi_{\lceil k/2\rceil}|}{2} \right\rceil + d_{r-1}(\chi_{\lceil k/2\rceil},t'). \tag{5.4}$$

Let ℓ be the highest level in $\chi_{\lceil k/2\rceil -1}$ if $\lceil \frac{k}{2} \rceil > 1$, or let ℓ be 0 otherwise. Clearly, $\ell \leq level_{t'}(x) - 2$ since $\chi_{\lceil k/2\rceil}$ consists of tasks from at least one level. Recall that $U(x,t',\ell)$ denotes the set of tasks in $I(x,t')$ that are on a level higher than ℓ relative to t'. The set $U(x,t',\ell)$ either equals $\chi_{\lceil k/2\rceil}$ or $\chi_{\lceil k/2\rceil} \cup \{w_{\lceil k/2\rceil}\}$, because x is maximal in $\chi_{\lceil k/2\rceil+1}$ and ℓ is the highest level below the lowest level contained in $\chi_{\lceil k/2\rceil}$. The only other task that might be contained in $U(x,t',\ell)$ (at first glance) is $w_{\lceil k/2\rceil+1}$. But since x is not a predecessor of $w_{\lceil k/2\rceil+1}$, by Definition 5.18.1, this is not the case. We claim that

$$d_{r-1}(x,t') = \left\lceil \frac{|F(\ell)|}{2} \right\rceil + d_{r-1}(F(\ell),t') \tag{5.5}$$

and either the size of $F(\ell)$ is even or $F(\ell)$ equals $\chi_{\lceil k/2\rceil}$. In the following we denote by M the value of the local variable M computed by the distance algorithm in iteration r for the tasks x and t' and level ℓ, *i.e.*, M is the set of maximal tasks in $U(x,t',\ell)$ that have minimum approximated distance to t' among the maximal tasks of $U(x,t',\ell)$ after iteration $r-1$. We have to consider three cases.

case 1: $U(x,t',\ell) = \chi_{\lceil k/2\rceil}$. Then the size of $U(x,t',\ell)$ is odd and therefore $F(\ell) = \chi_{\lceil k/2\rceil}$. We replace $\chi_{\lceil k/2\rceil}$ by $F(\ell)$ in (5.4) and obtain that (5.5) holds.

case 2: $U(x,t',\ell) = \chi_{\lceil k/2\rceil} \cup \{w_{\lceil k/2\rceil}\}$ and $|M| > 1$. Then $F(\ell) = \chi_{\lceil k/2\rceil} \cup \{w_{\lceil k/2\rceil}\}$ and its size is even. Clearly, $\lceil |F(\ell)|/2 \rceil = \lceil |\chi_{\lceil k/2\rceil}|/2 \rceil$. If M does not contain $w_{\lceil k/2\rceil}$, then $d_{r-1}(w_{\lceil k/2\rceil},t') > d_{r-1}(\chi_{\lceil k/2\rceil},t')$ since $w_{\lceil k/2\rceil}$ is maximal in $U(x,t',\ell)$. If M contains $w_{\lceil k/2\rceil}$, then it contains at least one task $u \in \max(\chi_{\lceil k/2\rceil})$ such that $d_{r-1}(w_{\lceil k/2\rceil},t') = d_{r-1}(u,t')$ since M contains at least two tasks. In either case, it holds that $d_{r-1}(F(\ell),t') = d_{r-1}(\chi_{\lceil k/2\rceil},t')$. Hence, we can replace $\chi_{\lceil k/2\rceil}$ by $F(\ell)$ in (5.4) and obtain that (5.5) holds.

case 3: $U(x,t',\ell) = \chi_{\lceil k/2\rceil} \cup \{w_{\lceil k/2\rceil}\}$ and $|M| = 1$. Then the size of $U(x,t',\ell)$ is even and therefore $F(\ell) = U(x,t',\ell) - M$. Clearly, $D(w_{\lceil k/2\rceil},t') \leq D(t,t') - S(w_{\lceil k/2\rceil})$. On the other hand,

$$D(v_{\lceil k/2\rceil},t') = \sum_{i=1}^{\lceil k/2\rceil -1} \left\lceil \frac{|\chi_i|}{2} \right\rceil = D(t,t') - S(v_{\lceil k/2\rceil}).$$

Since $w_{\lceil k/2\rceil}$ and $v_{\lceil k/2\rceil}$ are scheduled in the same timestep, we obtain that $D(w_{\lceil k/2\rceil},t') \leq D(v_{\lceil k/2\rceil},t')$. It follows that $d_{r-1}(w_{\lceil k/2\rceil},t') \leq D(\chi_{\lceil k/2\rceil},t')$, since $v_{\lceil k/2\rceil}$ has minimum distance to t' among tasks in $\chi_{\lceil k/2\rceil}$ and $d_{r-1}(w_{\lceil k/2\rceil},t') \leq D(w_{\lceil k/2\rceil},t')$, according to Lemma 5.23. Using (5.2) we obtain $d_{r-1}(w_{\lceil k/2\rceil},t') \leq d_{r-1}(\chi_{\lceil k/2\rceil},t')$. Since $w_{\lceil k/2\rceil}$ is maximal in $U(x,t',\ell)$ it follows that M contains $w_{\lceil k/2\rceil}$. Since we assumed that M consists of one task only we obtain that

$M = \{w_{\lceil k/2 \rceil}\}$. As a consequence, $F(\ell) = \chi_{\lceil k/2 \rceil}$. Again we replace $\chi_{\lceil k/2 \rceil}$ by $F(\ell)$ in (5.4) to prove that (5.5) holds.

Since y is minimal in $\chi_{\lceil k/2 \rceil}$, it has minimum distance to t among tasks in $\chi_{\lceil k/2 \rceil}$, *i.e.*, $D(t, \chi_{\lceil k/2 \rceil}) = D(t,y)$. Using (5.3) and the fact that $d_{r-1}(t,y) = D(t,y)$, we obtain $d_{r-1}(t, \chi_{\lceil k/2 \rceil}) = d_{r-1}(t,y)$. Hence,

$$d_{r-1}(t,A) \geq d_{r-1}(t,y), \qquad \forall A \neq \emptyset \text{ with } \min(A) \subseteq \min(\chi_{\lceil k/2 \rceil}). \tag{5.6}$$

Since equation (5.5) holds, we have $H(x,t') = F(\ell')$ for some level $\ell' \geq \ell$ where $d_{r-1}(x,t')$ equals $\lceil |F(\ell')| / 2 \rceil + d_{r-1}(F(\ell'),t')$. Because $\chi_{\lceil k/2 \rceil} \neq \emptyset$ it holds that $d_{r-1}(x,t') = D(x,t') > 0$. It follows that $H(x,t') \neq \emptyset$, which proves part one of the lemma.

If $H(x,t') = F(\ell)$, then either the size of $H(x,t')$ is even or $H(x,t')$ equals $\chi_{\lceil k/2 \rceil}$, as we have shown above. If the size of $H(x,t')$ is not even, then we can replace A by $H(x,t')$ in (5.6), and thus obtain that $d_{r-1}(t,H(x,t')) \geq d_{r-1}(t,y)$.

Otherwise $H(x,t') = F(\ell')$ for $\ell' > \ell$. The level of $w_{\lceil k/2 \rceil}$ is not higher than $\ell + 1$ since $w_{\lceil k/2 \rceil}$ is not on a higher level than $v_{\lceil k/2 \rceil}$ (Definition 5.18.3), and that task is on level $\ell + 1$. It follows that $w_{\lceil k/2 \rceil}$ is not contained in $U(x,t',\ell')$. As a consequence, $U(x,t',\ell')$ contains only tasks from $\chi_{\lceil k/2 \rceil}$, in particular, it contains all minimal tasks of $\chi_{\lceil k/2 \rceil}$ that are on a level $> \ell'$. Since $U(x,t',\ell')$ contains no other minimal tasks it holds that $\min(U(x,t',\ell')) \subseteq \min(\chi_{\lceil k/2 \rceil})$. The latter holds for $F(\ell')$ too, since the only task that we have possibly removed from $U(x,t',\ell')$ to obtain $F(\ell')$ is a maximal task of $U(x,t',\ell')$. Hence, $\min(H(x,t')) \subseteq \min(\chi_{\lceil k/2 \rceil})$. Again we replace A by $H(x,t')$ in (5.6) to obtain that $d_{r-1}(t,H(x,t')) \geq d_{r-1}(t,y)$, which proves part two of the lemma. □

In the last auxiliary lemma we show that the algorithm is able to determine an upper bound for the scheduling distance using a critical block in the block decomposition, provided that the distances between any task in the critical block and t, respectively t', are already known.

Lemma 5.25 *Let $\chi_k, \ldots, \chi_1$ be a canonical block decomposition for the task system $(I(t,t'), \prec)$. Let χ_j be a critical block with tasks from level ℓ. For all $x \in \chi_j$, let $d_{r-1}(t,x) = D(t,x)$, and let $d_{r-1}(x,t') = D(x,t')$. Then $a(\ell) \geq D(t,t')$.*

Proof. As in Figure 5.7, let p denote $\min_2\{d_{r-1}(t,y) \,|\, y \in \mathit{Crit}(t,t',\ell)\}$ and let q denote $\min_2\{d_{r-1}(y,t') \,|\, y \in \mathit{Crit}(t,t',\ell)\}$. All tasks of χ_j have the same distance e to t. By Definition 5.18.1, w_j has no predecessor in χ_j, for $j > 1$. The same holds for $j = 1$ too, since all tasks in χ_j are on level 1 in this case. As a consequence, all tasks of χ_j have the same distance f to t'. By Lemma 5.20, if a critical task from level ℓ is not contained in χ_j, then this task is w_j. In particular, at most one critical task from level ℓ is not contained in χ_j. Hence, there are two cases to consider.

case 1: All critical tasks z from level ℓ are contained in χ_j. Then for all of them it holds that $d_{r-1}(t,z) = D(t,z) = e$ and $d_{r-1}(z,t') = D(z,t') = f$. Since there is at least one critical task on level ℓ we obtain that $\{d_{r-1}(t,y)\,|\,y \in \mathit{Crit}(t,t',\ell)\} = \{e\}$ and $\{d_{r-1}(y,t')\,|\,y \in \mathit{Crit}(t,t',\ell)\} = \{f\}$. Hence, $p = e$ and $q = f$.

case 2: w_j is a critical task on level ℓ but all other critical tasks from level ℓ are contained in χ_j. It holds that $D(w_j,t') \leq f$ and $D(t,w_j) \leq e$. By Lemma 5.23, $d_{r-1}(t,w_j)$ is a lower bound for $D(t,w_j)$ and $d_{r-1}(w_j,t')$ is one for $D(w_j,t')$. As a consequence, $d_{r-1}(t,w_j) \leq e$ and $d_{r-1}(w_j,t') \leq f$. For all critical tasks z other than w_j it holds that $d_{r-1}(t,z) = D(t,z) = e$ and $d_{r-1}(z,t') = D(z,t') = f$. By Lemma 5.21, all tasks in χ_j are critical. Since χ_j is not empty, there is at least one critical task in χ_j. We derive that

$$\{d_{r-1}(t,y)\,|\,y \in \mathit{Crit}(t,t',\ell)\} = \{d_{r-1}(t,w_j), e\}$$

and

$$\{d_{r-1}(y,t')\,|\,y \in \mathit{Crit}(t,t',\ell)\} = \{d_{r-1}(w_j,t'), f\}.$$

Since $d_{r-1}(t,w_j) \leq e$ and $d_{r-1}(w_j,t') \leq f$ it follows that $p = e$ and $q = f$.

It follows that either $C = \chi_j$ or $C = \chi_j \cup \{w_j\}$. In either case, the approximated distance of t to any task in C is e and the approximated distance of any task in C to t' is f. Moreover, $\lceil |\chi_j|/2 \rceil = \lceil |C|/2 \rceil$ since the size of χ_j is odd. We obtain

$$a(\ell) = d_{r-1}(t,C) + d_{r-1}(C,t') + \left\lceil \frac{|C|}{2} \right\rceil = e + f + \left\lceil \frac{|\chi_j|}{2} \right\rceil.$$

Since every $x \in \chi_j$ is successor of all tasks in $\chi_k, \ldots, \chi_{j+1}$ and predecessor of all tasks in $\chi_{j-1}, \ldots, \chi_1$, we require at least $\sum_{i=j+1}^{k} \lceil |\chi_i|/2 \rceil$ timesteps to schedule $I(t,x)$ and at least $\sum_{i=1}^{j-1} \lceil |\chi_i|/2 \rceil$ timesteps to schedule $I(x,t')$. Hence,

$$e + f + \left\lceil \frac{|\chi_j|}{2} \right\rceil \geq \sum_{i=j+1}^{k} \left\lceil \frac{|\chi_i|}{2} \right\rceil + \sum_{i=1}^{j-1} \left\lceil \frac{|\chi_i|}{2} \right\rceil + \left\lceil \frac{|\chi_j|}{2} \right\rceil = D(t,t').$$

We conclude that $a(\ell) \geq D(t,t')$. □

We have now everything in place to prove that the approximated distances eventually converge to the correct values.

Lemma 5.26 *Let $\chi_k, \ldots, \chi_1$ be a canonical block decomposition for the task system $(I(t,t'), \prec)$ such that $k \leq 2^r$. Then $d_r(t,t') = D(t,t')$.*

Proof. By induction on r. The lemma holds for $r = 0$ because if there is only one block, then $D(t,t') = \lceil |I(t,t')|/2 \rceil = d_0(t,t')$. Let it hold for $r-1 \geq 0$. Since $\left\lceil \frac{k}{2} \right\rceil \leq 2^{r-1}$ we can assume that the lemma holds for all pairs of tasks with a canonical block decomposition consisting of at most $\left\lceil \frac{k}{2} \right\rceil$ blocks. Let t_1, ..., $t_{level_{t'}(t)-1}$ be a longest path in $(I(t,t'), \prec)$ such that t_i is on level i (relative to t') and let ℓ be the highest level in $\chi_{\lceil k/2 \rceil}$.

We first consider the case where $t_\ell \in \chi_{\lceil k/2 \rceil}$ and $t_{\ell+1} \in \chi_{\lceil k/2 \rceil+1}$. Clearly, t_ℓ is minimal in $\chi_{\lceil k/2 \rceil}$ and $t_{\ell+1}$ is maximal in $\chi_{\lceil k/2 \rceil+1}$. By applying Lemma 5.24 we obtain that $H(t_{\ell+1},t') \neq \emptyset$ and furthermore the size of $H(t_{\ell+1},t')$ is even or $d_{r-1}(t,H(t_{\ell+1},t')) \geq d_{r-1}(t,t_\ell)$. As a consequence, the distance algorithm assigns $d_{r-1}(t,t_\ell) + d_{r-1}(t_{\ell+1},t')$ to $b(\ell)$. According to Lemma 5.22, there exist canonical block decompositions for $(I(t_{\ell+1},t'), \prec)$ and $(I(t,t_\ell), \prec)$ consisting of at most $\left\lceil \frac{k}{2} \right\rceil$ blocks each. By inductive hypothesis, $b(\ell) = D(t,t_\ell) + D(t_{\ell+1},t')$, and that is at least $D(t,t')$ according to Lemma 5.16.

Next, we consider the cases where either $t_{\ell+1} \notin \chi_{\lceil k/2 \rceil+1}$ or $t_\ell \notin \chi_{\lceil k/2 \rceil}$. We shall see that in the former case $\chi_{\lceil k/2 \rceil+1}$ is a critical block, while in the latter case $\chi_{\lceil k/2 \rceil}$ is a critical block.

case 1: $t_{\ell+1} \notin \chi_{\lceil k/2 \rceil+1}$. Clearly, $\chi_{\lceil k/2 \rceil+1}$ contains at least one task from level $\ell+1$, since ℓ is the highest level in $\chi_{\lceil k/2 \rceil}$. By Lemma 5.20, $t_{\ell+1} = w_{\lceil k/2 \rceil+1}$. This implies that $\chi_{\lceil k/2 \rceil+1}$ is a critical block (Definition 5.18.2).

case 2: $t_\ell \notin \chi_{\lceil k/2 \rceil}$. Block $\chi_{\lceil k/2 \rceil}$ contains at least one task on level ℓ. We obtain from Lemma 5.20 that $t_\ell = w_{\lceil k/2 \rceil}$. If $\left\lceil \frac{k}{2} \right\rceil = 1$ then t_ℓ is on level 1. Hence $\ell = 1$. It follows that $\chi_{\lceil k/2 \rceil}$ is critical since ℓ is the highest level in $\chi_{\lceil k/2 \rceil}$ and in this case the only level. Otherwise $\left\lceil \frac{k}{2} \right\rceil > 1$ and we can apply Definition 5.18.2, from which we obtain that $\chi_{\lceil k/2 \rceil}$ is a critical block.

Now, let χ be either $\chi_{\lceil k/2 \rceil+1}$ or $\chi_{\lceil k/2 \rceil}$, whichever of them is critical, and let ℓ' be the level of tasks in χ. Let $x \in \chi$. By Lemma 5.22, there exist canonical block decompositions for $(I(x,t'), \prec)$ and $(I(t,x), \prec)$ consisting of at most $\left\lceil \frac{k}{2} \right\rceil$ blocks each. By inductive hypothesis, $d_{r-1}(t,x) = D(t,x)$ and $d_{r-1}(x,t') = D(x,t')$. By applying Lemma 5.25 we obtain that $a(\ell') \geq D(t,t')$.

We conclude that $b(\ell)$ or $a(\ell+1)$ or $a(\ell)$ is at least $D(t,t')$. Since the maximum of all $b(\star)$'s and $a(\star)$'s is assigned to $d_r(t,t')$, we obtain that $d_r(t,t') \geq D(t,t')$. On the other hand, Lemma 5.23 shows that $d_r(t,t')$ does not exceed $D(t,t')$. Hence $d_r(t,t') = D(t,t')$. □

Theorem 5.27 *The distance algorithm in Figure 5.7 correctly computes the* *scheduling distances $D(t,t')$ for all tasks t and t'.*

Proof. Let $\chi_k,\ldots,\chi_1$ be a canonical block decomposition for $(I(t,t'),\prec)$. Such a decomposition exists according to Lemma 5.19. Since $k \leq n$ it holds that $k \leq 2^{\lceil \log n \rceil}$. We apply Lemma 5.26 and obtain that $d_{\lceil \log n \rceil}(t,t') = D(t,t')$. □

Theorem 5.28 *The distance algorithm can be implemented to run in $O(\log^2 n)$ time using $n^3/\log n$ processors of a CREW PRAM.*

Proof. We assume that the precedence graph is given as an adjacency matrix and each task is represented by a natural number, *i.e.*, $T = \{1,\ldots,n\}$. In the preprocessing phase we perform the following steps.

(1) Select longest paths: For each pair of tasks t and t', we have to extract from the precedence graph a longest path from t to t'. According to Theorem 3.29, this can be done for all pairs of tasks simultaneously in time $O(\log^2 n)$ using $n^3/\log n$ EREW PRAM processors. At the same time, this algorithm computes an all-pairs longest path matrix B. The matrix B is an $n \times n$ matrix such that $B(t,t')$ is the length of a longest path between t and t' or $-\infty$ if no path exists. Note that the set $I(t,t')$ consists of all tasks x such that $B(t,x) > 0$ and $B(x,t') > 0$.

(2) Determine critical tasks: A task $x \in I(t,t')$ is critical in $I(t,t')$ iff $B(t,x) + B(x,t') = B(t,t')$. Hence, we can decide in constant sequential time whether a task is critical.

(3) Initialize approximated distances: To compute $d_0(t,t')$ we require the size of $I(t,t')$. This information can easily be obtained using prefix operations.

After preprocessing, the main loop is entered. Before we start with the computations listed in Figure 5.7, we sort the tasks of each set $I(t,t')$ lexicographically by level and by the current approximated distances to t':

(4) Sort $I(t,t')$ in level order: Let $X(t,t')$ be the sequence that consists of $I(t,t')$ sorted by nondecreasing level such that tasks on the same level are sorted by their approximated distances to t'. To compute $X(t,t')$ for all pairs of tasks t and t', we proceed as follows. For each t', we sort all tasks in nondecreasing order of their level relative to t' such that tasks on the same level are sorted by their current approximated distance to t' (the level of x relative to t' is $B(x,t')$). For each task t, the sequence obtained contains a subsequence that equals $X(t,t')$. This subsequence can easily be extracted using a prefix operation. To perform these operations for all pairs of tasks in parallel, we require $O(\log n)$ time on $n^3/\log n$ processors, since we first sort n sequences of size n and then we extract n subsequences from each of the n sorted sequences.

(5) Compute the sets $H(t,t')$: Let ℓ be a level in $\{0,\ldots,level_{t'}(t)-2\}$. Clearly, the maximal tasks of $U(t,t',\ell)$ are on level $\ell+1$ relative to t'. Hence, to determine whether $|M|=1$, it suffices to check the approximated distances between the tasks on level $\ell+1$ and t'. Let $x_1,\ldots,x_m$ denote the sequence $X(t,t')$. The last $|U(t,t',\ell)|$ tasks in $X(t,t')$ are precisely the tasks of $U(t,t',\ell)$. Let x_j be the first task in $X(t,t')$ that is on level $\ell+1$. It holds that $|M|=1$ iff the number of tasks on level $\ell+1$ in $U(t,t',\ell)$ is 1 or their number is greater than 1 but $d_{r-1}(x_{j+1},t') > d_{r-1}(x_j,t')$. Moreover, if $|M|=1$ then $M=\{x_j\}$. Hence, we can decide in constant sequential time whether $|M|=1$, and if that is the case, we can easily determine M. If $|M|=1$ and $|U(t,t',\ell)|$ is even, then the last $|U(t,t',\ell)|-1$ tasks in $X(t,t')$ form the set $F(\ell)$. Otherwise, $F(\ell)$ consists of the last $|U(t,t',\ell)|$ tasks in $X(t,t')$. Therefore,

$$d_{r-1}(F(\ell),t') = \begin{cases} d_{r-1}(x_j,t') & \text{if } F(\ell)=U(t,t',\ell),\\ d_{r-1}(x_{j+1},t') & \text{otherwise.}\end{cases}$$

We use a prefix operation to determine the highest level ℓ' such that $d_{r-1}(t,t') = \lceil |F(\ell')|/2\rceil + d_{r-1}(F(\ell'),t')$. In the end, we assign $F(\ell')$ to $H(t,t')$. In fact, we only have to keep the minimal tasks of $H(t,t')$ and its size. All of the above operations can be performed in time $O(\log n)$ using $n/\log n$ processors. Since there are $O(n^2)$ pairs of tasks, we can determine all sets $H(t,t')$ simultaneously using $n^3/\log n$ processors.

(6) Determine approximations based on critical blocks: To determine $a(\ell)$ for a level ℓ in $I(t,t')$, we only have to consider tasks on level ℓ. Therefore, to compute the approximations for all levels in $I(t,t')$, we can use prefix operations segmented by the levels in $I(t,t')$. Since the number of tasks in $I(t,t')$ is $O(n)$, such a segmented prefix operation takes $O(\log n)$ time on $n/\log n$ processors. Since there are $O(n^2)$ pairs of tasks, we can compute all approximations based on critical blocks in time $O(\log n)$ using $n^3/\log n$ processors.

(7) Determine approximations based on critical tasks: Let ℓ denote a level between 1 and $level_{t'}(t)-2$, and let t_1, …, $t_{level_{t'}(t)-1}$ be the longest path in $(I(t,t'),\prec)$ chosen by the algorithm, where t_i is on level i relative to t'. To determine $d_{r-1}(t,H(t_{\ell+1},t'))$, we have to look at the minimal tasks in $H(t_{\ell+1},t')$. More precisely, we have to perform a prefix-minima operation on the approximated distances between t and the minimal tasks of $H(t_{\ell+1},t')$. Note that for any two tasks t_i and t_j, the minimal tasks of $H(t_i,t')$ and the minimal tasks of $H(t_j,t')$ are disjoint. This is because $t_1,\ldots,t_{level_{t'}(t)-1}$ is a longest path in $I(t,t')$. Therefore, the total number of tasks that we have to consider in order to determine $d_{r-1}(t,H(t_{\ell+1},t'))$ for all levels ℓ in $I(t,t')$, is at most the size of $I(t,t')$, which is $O(n)$. Hence, we can perform all $level_{t'}(t)-2$ prefix-minima opera-

tions in time $O(\log n)$ on $n / \log n$ processors. Clearly, when $d_{r-1}(t, H(t_{\ell+1}, t'))$ is determined, the computation of $b(\ell)$ takes only constant time. Since there are $O(n^2)$ pairs of tasks (t, t') that we have to consider, the computation of all approximations $b(\ell)$ requires $n^3 / \log n$ processors. During these computations, simultaneous read accesses to the sets $H(t, t')$ may occur. Hence, we require the CREW PRAM.

(8) Compute new distance approximations: For each pair of tasks (t, t'), a constant number of prefix-maxima operations, each on at most n elements, suffice to compute $d_r(t, t')$. Hence, the time required to determine the new distance approximations for all pairs of tasks is $O(\log n)$ and $n^3 / \log n$ processors are used.

The main loop of the distance algorithm is iterated $\lceil \log n \rceil$ times. As we have seen above, each iteration takes $O(\log n)$ time. As a consequence, the total time spent in the main loop is $O(\log^2 n)$ if $n^3 / \log n$ processors are used. Since preprocessing can be performed within the same bounds, the theorem follows. □

5.7 Computing the Jump Sequence

In the sequel we assume that T contains a sufficient number of empty tasks $e_1, e_2, \ldots$ on level 0 that can be used for jumps to level 0 if necessary. For instance, if the number of levels that jump is k, then we add k empty tasks.

Recall that in LMJ schedules only solitary tasks are used to jump from levels (cf. Lemma 5.4). Using the construction of Section 5.3 and the distance algorithm, we are able to determine which levels jump and which tasks are possibly used to jump from these levels. We are now interested in the tasks that are used as fill-ins. A necessary condition for a task x to be a fill-in for the jump from level ℓ is that there exists a solitary task y on level ℓ that is independent of x. In the following we introduce another necessary condition. Let $high(x)$ be the highest level ℓ such that on every jumping level ℓ' with $level(x) < \ell' \leq \ell$ there exists a solitary task that is independent of x. If no such level exists, then let $high(x) = 0$. If x is an empty task, then $high(x)$ is the highest jumping level since an empty task can be paired with all solitary tasks.

Lemma 5.29 *Let ℓ be a level higher than $high(x)$. Then there exists no LMJ schedule where ℓ jumps to x.*

Proof. If $high(x) = 0$ then the lemma clearly holds. Otherwise, assume the contrary and let ℓ jump to x in some LMJ schedule. Let y be the task on level ℓ

that is paired with x. Clearly, y is a solitary task (Lemma 5.4). According to the definition of $\mathit{high}(x)$, there must exist a jumping level ℓ' with $\mathit{high}(x) < \ell' < \ell$ such that no solitary task on level ℓ' is independent of x. In particular, the solitary task z that is used to jump from level ℓ' must be a predecessor of x. But since x is paired with y and y is scheduled earlier than z, this violates the precedence constraints, a contradiction. We conclude that in no LMJ schedule level ℓ jumps to x. □

Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be levels that jump. A *landing set* for $\{\ell_1, \ldots, \ell_k\}$ is an ordered set of (possibly empty) tasks $\{t_1, \ldots, t_k\}$ such that $\mathit{level}(t_i) < \ell_i \leq \mathit{high}(t_i)$, for $i = 1, \ldots, k$. A landing set $\{t_1, \ldots, t_k\}$ *implies* the jump sequence $\mathit{level}(t_1), \ldots, \mathit{level}(t_k)$. Landing sets provide us with candidates for the fill-ins used in an LMJ schedule. Not every landing set is a set of fill-ins for an LMJ schedule but the converse is true. Let $t_1, \ldots, t_k$ be tasks that are used as fill-ins in an LMJ schedule. Then, according to Lemma 5.29, it holds for every fill-in t_i that $\ell_i \leq \mathit{high}(t_i)$. It is furthermore clear that $\mathit{level}(t_i) < \ell_i$. It follows that $\{t_1, \ldots, t_k\}$ is a landing set for $\{\ell_1, \ldots, \ell_k\}$.

In the rest of this section, we are concerned with determining a landing set that does not contain any of a set of reserved solitary tasks but nevertheless implies the LMJ sequence. We start with some observations on landing sets.

Lemma 5.30 *Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be levels that jump to some level ℓ in an LMJ schedule for $(T, \prec)$ and let t be an arbitrary solitary task on level ℓ. Then there exists a landing set for $\{\ell_1, \ldots, \ell_k\}$ on level ℓ that does not contain t.*

Proof. Assume the contrary, *i.e.*, every possible landing set for $\{\ell_1, \ldots, \ell_k\}$ on level ℓ contains t. If we restrict an LMJ schedule S for $(T, \prec)$ to the tasks on levels $L, \ldots, \ell$, then the result is an LMJ schedule for those tasks (Lemma 5.2). Since $\ell_1, \ldots, \ell_k$ jump to level ℓ in S, it holds that $\ell_1, \ldots, \ell_k$ jump to level ℓ in an LMJ schedule for the tasks on levels $L, \ldots, \ell$. Since the fill-in tasks used in an LMJ schedule form a landing set, as we have observed before, and every landing set for $\{\ell_1, \ldots, \ell_k\}$ on level ℓ contains t, there is no LMJ schedule for tasks on levels $L, \ldots, \ell$ where t is the only task mapped to the last timestep. Hence, t is not solitary. A contradiction. We conclude that at least one landing set for $\{\ell_1, \ldots, \ell_k\}$ exists on level ℓ that does not contain t. □

Lemma 5.31 *Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be the levels that jump in an LMJ schedule for $(T, \prec)$. Let R be a set of solitary tasks, one from each level that jumps. Then there exists a landing set for $\{\ell_1, \ldots, \ell_k\}$ that does not contain tasks from R but implies the LMJ sequence.*

Proof. Let $\{t_1,\dots,t_k\}$ be the landing set used in an LMJ schedule for $(T,\prec)$. Let W be the set of levels jumped to, and let J_ℓ be the levels that jump to level $\ell \in W$. The landing set $\{t_1,\dots,t_k\}$ consists of disjoint subsets T_ℓ, one for each $\ell \in W$, such that T_ℓ is a landing set for J_ℓ. Let $t \in R$ be the reserved solitary task on level ℓ. According to Lemma 5.30, there exists an alternative landing set T'_ℓ for J_ℓ on level ℓ that does not contain t. For each $\ell \in W$, replace each task $t_i \in T_\ell$ in $\{t_1,\dots,t_k\}$ with its equivalent t'_i from T'_ℓ. We obtain a landing set $\{t'_1,\dots,t'_k\}$ $(= \bigcup_{\ell \in W} T'_\ell)$ for $\{\ell_1,\dots,\ell_k\}$ that does not contain any task of R but $\mathit{level}(t_i) = \mathit{level}(t'_i)$, for $i = 1,\dots,k$. □

It remains to show how such a landing set can be computed. The following lemma suggests a greedy approach. Any landing set that is constructed in a greedy "highest level first" fashion, among the tasks that are not reserved, implies a jump sequence that is lexicographically greater than or equal to the LMJ sequence. As it turns out in the next section, the implied jump sequence actually *is* the LMJ sequence. The proof of the latter is deferred for technical reasons.

Lemma 5.32 *Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be the levels that jump in an LMJ schedule for $(T,\prec)$. Let R be a set of solitary tasks, one from each level that jumps. Moreover, let $\{t_1,\dots,t_k\}$ be a landing set for $\{\ell_1,\dots,\ell_k\}$ such that t_i is a highest task in $T - R - \{t_1,\dots,t_{i-1}\}$ with $\mathit{level}(t_i) < \ell_i \le \mathit{high}(t_i)$, for $i = 1,\dots,k$. Then the jump sequence implied by $\{t_1,\dots,t_k\}$ is lexicographically greater than or equal to the LMJ sequence of $(T,\prec)$.*

Proof. Let $\{t'_1,\dots,t'_k\}$ be a landing set that contains no task of R but implies the LMJ sequence. Such a landing set exists according to Lemma 5.31. Assume that the jump sequence implied by $\{t_1,\dots,t_k\}$ is lexicographically less than the LMJ sequence. Then there exists an index j such that $\mathit{level}(t_j) < \mathit{level}(t'_j)$ and $\mathit{level}(t_i) = \mathit{level}(t'_i)$ for $i < j$. Let ℓ denote the level of t'_j, and let X be the subset of $\{t'_1,\dots,t'_j\}$ that is on level ℓ. The tasks in X form a landing set for some of the levels in $\{\ell_1,\dots,\ell_j\}$, all of which are at least as high as ℓ_j. Therefore, the high values of tasks in X are at least as high as ℓ_j. Hence, for all $x \in X$ it holds that $\mathit{level}(x) < \ell_j \le \mathit{high}(x)$. The number of tasks in $\{t_1,\dots,t_{j-1}\}$ on level ℓ equals $|X| - 1$ since $t'_j \in X$ and $\mathit{level}(t_i) = \mathit{level}(t'_i)$ for $i < j$. Because X is contained in $T - R$ there must be at least one task $x \in X$ that is also contained in $T - R - \{t_1,\dots,t_{j-1}\}$. Therefore, x is a task in $T - R - \{t_1,\dots,t_{j-1}\}$ with $\mathit{level}(x) < \ell_j \le \mathit{high}(x)$ and x is higher than t_j. A contradiction to the choice of t_j. Hence, our assumption is wrong and the jump sequence implied by

$\{t_1,\ldots,t_k\}$ is lexicographically greater than or equal to the LMJ sequence of $(T,\prec)$. □

In the following we show how such a landing set can be obtained from a maximum matching in a convex bipartite graph constructed from the precedence graph (cf. Section 3.13).

Lemma 5.33 *Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be the levels that jump in an LMJ schedule for $(T,\prec)$. Let R be a set of solitary tasks, one from each level that jumps. Then we can compute a landing set X for $\{\ell_1,\ldots,\ell_k\}$ in time $O(\log^2 n)$ using $n^2/\log n$ processors of a CREW PRAM such that X and R are disjoint and X implies a jump sequence that is lexicographically greater than or equal to the LMJ sequence of $(T,\prec)$.*

Proof. Let (A,B,E) be the bipartite graph defined as follows. Let $\ell'_1 > \ell'_2 > \cdots > \ell'_{L-k}$ be the levels in $\{L,\ldots,1\}$ that do not jump. The set $A = \{L,\ldots,1\}$ consists of all levels, ordered by decreasing level. We assume that the elements of A are numbered in this order from 1 to n, *e.g.*, level L has number 1. The set $B = \{t_1,\ldots,t_n\}$ consists of all tasks $t \in T - R$ such that $high(t) > level(t)$ (recall that T contains k empty tasks) plus $L-k$ dummy tasks $c_1,\ldots,c_{L-k}$. For each $\ell \in A$ and each nondummy task $t \in B$, we add an edge (ℓ,t) to E iff $level(t) < \ell \leq high(t)$. For each dummy task c_i, we add an edge (ℓ'_i,c_i) to E. The resulting bipartite graph is convex on A since if $(\ell,t) \in E$ and $(\ell',t) \in E$ then $(\ell'',t) \in E$ for all levels ℓ'' between ℓ and ℓ', due to the definition of $high(t)$. The first vertex in A connected to a nondummy task $t \in B$ is $high(t)$ and the last vertex is $level(t)+1$. The only vertex in A connected to a dummy task c_i is ℓ'_i. For every task $t \in B$, let $begin(t)$ denote the number of the first element of A connected to t, and let $end(t)$ denote the number of the last element of A connected to t. Note that for every two nondummy tasks x and y in B it holds that $level(x) > level(y)$ iff $end(x) < end(y)$. Once the high-values are determined, it is easy to construct (A,B,E) in parallel, representing it as a sequence of tuples $(begin(t_1),end(t_1))$, …, $(begin(t_n),end(t_n))$.

To determine the high-value of a task t, we proceed as follows. First, for each jumping level higher than $level(t)$, we count the number of solitary tasks that are independent of t. This is performed using a prefix-sums operation segmented by levels. Next, for each jumping level ℓ, we count the number of jumping levels ℓ' with $level(t) < \ell' \leq \ell$ that contain at least one solitary task independent of t. Let x_ℓ denote this number. Similarly, we count for each jumping level ℓ the number of jumping levels ℓ' with $level(t) < \ell' \leq \ell$. Let y_ℓ denote this number. Clearly, $high(t)$ is the highest jumping level ℓ such that $x_\ell = y_\ell$. We can determine the highest such level using a prefix-maxima operation. For each task, we require $O(\log n)$ time and $n / \log n$ processors to perform all necessary prefix operations. Since the high-values of all tasks can be computed in parallel, the total resource requirements are $O(\log n)$ time and $n^2 / \log n$ processors.

Next, we compute a maximum matching M in (A,B,E) using the algorithm of Dekel and Sahni [DS84] which requires $O(\log^2 n)$ time on $n / \log n$ processors (Theorem 3.30). Recall that this algorithm computes matchings according to Glover's strategy. Hence, levels are matched in decreasing order. Each level is matched with an unmatched task $t \in B$ such that $\mathit{end}(t)$ is as small as possible. We claim that every level that does not jump is either matched with its corresponding dummy task or with a nondummy task that would be unmatched otherwise. To prove the claim, let ℓ be the level currently under consideration, and let ℓ be the i-th level in A. Assume that ℓ does not jump but is matched with a nondummy task t. Since the dummy task c corresponding to ℓ is still unmatched and $\mathit{end}(c) = i$, it follows that $\mathit{end}(t) = i$ too. Consequently, no unmatched level other than ℓ can be matched with t. Since t has not been matched with a level higher than ℓ, we can derive that t would be unmatched if not matched to ℓ, which proves the claim. It follows that the algorithm matches jumping levels as if only jumping levels were present in A. As we have observed above, $\mathit{level}(x) > \mathit{level}(y)$ iff $\mathit{end}(x) < \mathit{end}(y)$, for every two nondummy tasks x and y in B. Hence, if the level ℓ currently under consideration is jumping, it will be matched with a highest nondummy task t not matched to higher jumping levels such that $\mathit{level}(t) < \ell \le \mathit{high}(t)$. Since there are k empty tasks in B that can be matched with any level that jumps, it holds that all levels that jump are matched in M. We remove all edges from M that match levels that do not jump, and thus obtain a matching $\{(\ell_1, t'_1), \ldots, (\ell_k, t'_k)\}$ where t'_i is a highest task in $T - R - \{t'_1, \ldots, t'_{i-1}\}$ such that $\mathit{level}(t'_i) < \ell_i \le \mathit{high}(t'_i)$, for $i = 1, \ldots, k$. By Lemma 5.32, the jump sequence implied by $\{t'_1, \ldots, t'_k\}$ is lexicographically greater than or equal to the LMJ sequence of $(T, \prec)$. □

In Figure 5.8, a convex bipartite graph for the task system in Figure 5.1 is given. As reserved solitary tasks, we have chosen $\{t_{15}, t_{12}, t_{11}, t_9, t_4\}$, one from each level that jumps. The edges shaded grey form a maximum matching obtained from the algorithm of Dekel and Sahni. We remove the edges $(6, c_1)$ and $(1, c_2)$, since levels 6 and 1 do not jump, and obtain the matching $\{(7, t_{10}), (5, e_1), (4, t_6), (3, t_5), (2, t_1)\}$, which corresponds to the landing set $\{t_{10}, e_1, t_6, t_5, t_1\}$. The jump sequence implied by this landing set is $3, 0, 3, 2, 1$, which is the LMJ sequence of the task system.

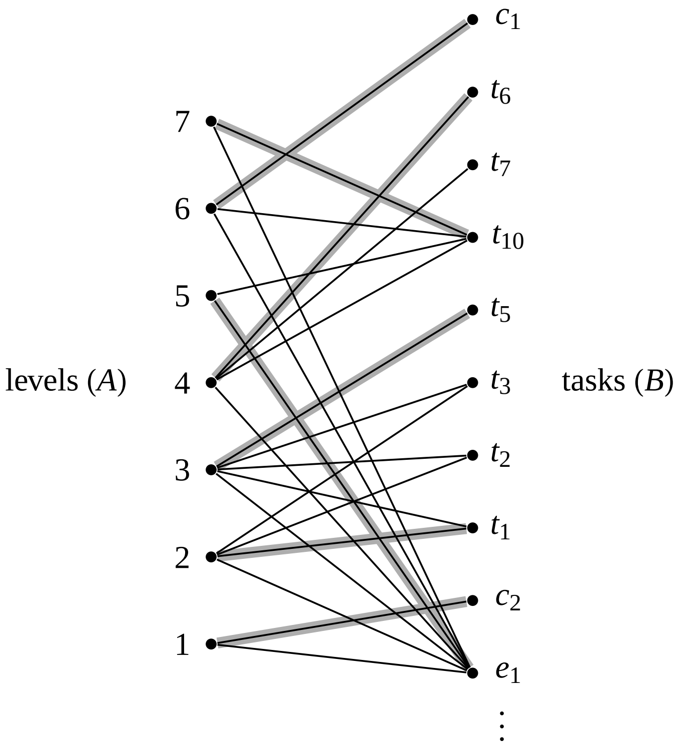


Figure 5.8: *Levels that jump are matched with tasks that can be used as fill-ins, where tasks* $\{t_{15}, t_{12}, t_{11}, t_9, t_4\}$ *have been chosen as reserved tasks. The landing set obtained from the maximum matching is* $\{t_{10}, e_1, t_6, t_5, t_1\}$.

5.8 Selecting the Actual Jumps

The remaining step in our scheduling algorithm is to determine the actual jumps of an LMJ schedule. Once the actual jumps are selected, it is easy to construct the schedule. As we have shown in the last section, we are able to determine a landing set for the levels that jump in an LMJ schedule that implies a jump sequence that is lexicographically greater than or equal to the LMJ sequence. Although the jump sequence implied by this landing set actually *is* the LMJ sequence, as we will prove in the sequel, it is possible that this landing set is not used by any LMJ schedule. This is because it might contain a solitary task that cannot be used as fill-in but rather has to be used to jump from some level. In this section, we are concerned with transforming the landing set from the previous section into a landing set that can be used in an LMJ schedule. To this end, we introduce the notion of *candidate pairs*. A candidate pair is simply a pair of tasks that fulfill all necessary conditions for being an actual jump in an LMJ schedule.

Definition 5.34 *A pair of tasks* (x,y) *is called a* candidate pair *if* x *is solitary,* $x \not\prec y$*, and level*$(y) <$ *level*$(x) \leq$ *high*(y)*.*

We will show that any set of disjoint candidate pairs, one pair for each level that jumps, forms the actual jumps of an LMJ schedule, provided that the implied jump sequence is lexicographically greater than or equal to the LMJ sequence. Such a set of candidate pairs is called a *candidate set.*

Definition 5.35 *Let* $\ell_1 > \ell_2 > \cdots > \ell_k$ *be the levels that jump in an LMJ schedule for* $(T,\prec)$*. A set of disjoint candidate pairs* $\{(t_1,t'_1),\ldots,(t_k,t'_k)\}$ *is called a* candidate set *for* $(T,\prec)$ *if*

1. *the level of* t_i *is* ℓ_i*, for* $i = 1,\ldots,k$*, and*
2. *the jump sequence implied by* $\{t'_1,\ldots,t'_k\}$ *is lexicographically greater than or equal to the LMJ sequence of* $(T,\prec)$*.*

Lemma 5.36 *Any candidate set for* $(T,\prec)$ *forms the actual jumps of an LMJ schedule.*

Proof. Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be the levels that jump, and let $\{t_1,\ldots,t_k\}$ be the landing set used in a candidate set P. Moreover, let $\ell'_1,\ldots,\ell'_k$ be the LMJ sequence. By definition, the jump sequence implied by $\{t_1,\ldots,t_k\}$ is lexicographically greater than or equal to the LMJ sequence. Let i be the first position where *level*$(t_i) > \ell'_i$ or let $i = k$ if the jump sequence implied by $\{t_1,\ldots,t_k\}$ is the LMJ sequence.

Consider the mapping S constructed as follows. We map tasks in the order of nonincreasing level to timesteps, and for the jumps from levels $\ell_1,\ldots,\ell_i$ we use the candidate pairs of P as actual jumps. This is possible since the candidate pairs in P are all disjoint and the jump sequence implied by $\{t_1,\ldots,t_{i-1}\}$ is a prefix of the LMJ sequence. The remaining tasks are scheduled using arbitrary available tasks as fill-ins for jumps.

Assume that S violates precedence constraints, and let s be the first timestep with such a violation. This first violation occurs not later than the jump from level ℓ_i, since behind the jump from level ℓ_i all remaining tasks are scheduled in proper level order fashion. Since precedence constraints can only be violated by jumps, s contains a candidate pair $(t, t_j) \in P$ for a jump (ℓ_j, ℓ) with $j \leq i$, and t_j is not available at timestep s. Because t and t_j are independent, there must exist a predecessor of t_j that is mapped to a timestep behind s. Let x be the earliest such task, *i.e.*, $x \prec t_j$, $S(x) > s$, and all predecessors of x are mapped to timesteps before s. Tasks t and x are independent, since otherwise t and t_j would not be independent and (t, t_j) could not be a candidate pair. Hence, t and x can both be mapped to timestep s. Since no violation of precedence constraints occurs before timestep s in S, we can construct a level schedule S' such that $\mathit{level}(t_1), \ldots, \mathit{level}(t_{j-1}), \mathit{level}(x)$ is a prefix of the jump sequence of S'. Since x is higher than t_j, it follows that level ℓ_j jumps higher in S' than proposed in the candidate pair (t, t_j). It follows that the jump sequence of S' is lexicographically greater than the jump sequence implied by $\{t_1, \ldots, t_k\}$ and hence lexicographically greater than the LMJ sequence, contradicting the fact that the LMJ sequence is the lexicographically greatest jump sequence of any level schedule. Hence, our assumption is wrong and S is a valid level schedule.

Clearly, $\mathit{level}(t_1), \ldots, \mathit{level}(t_i)$ is a prefix of the jump sequence of S. Assume that $\mathit{level}(t_i) > \ell'_i$. Then the jump sequence of S is lexicographically greater than the LMJ sequence, again a contradiction. Hence, our assumption is wrong and we conclude that $\mathit{level}(t_1), \ldots, \mathit{level}(t_k)$ is the LMJ sequence and S is an LMJ schedule with the candidate pairs P as actual jumps. □

Let $\ell_1 > \ell_2 > \cdots > \ell_k$ be the levels that jump in an LMJ schedule. To compute a candidate set we proceed as follows.

For $i = 1, \ldots, k$, choose a solitary task s_i on level ℓ_i with maximum high-value. Let R denote the set $\{s_1, \ldots, s_k\}$, and let $\{t_1, \ldots, t_k\}$ be a landing set for $\{\ell_1, \ldots, \ell_k\}$ such that R and $\{t_1, \ldots, t_k\}$ are disjoint and the jump sequence implied by $\{t_1, \ldots, t_k\}$ is lexicographically greater than or equal to the LMJ sequence. Such a landing set can be computed efficiently in parallel, as we have shown in Lemma 5.33. For $i = 1, \ldots, k$, choose a solitary task s'_i on level ℓ_i that is independent of t_i. The pair (s'_i, t_i) is called the *primary candidate pair* for the jump from level ℓ_i and we denote it by $\mathcal{P}(\ell_i)$.

The *secondary candidate pair* for the jump from level ℓ_i, denoted by $\mathcal{S}(\ell_i)$, is defined as follows. If t_i is not solitary, then the primary candidate pair for ℓ_i is used as secondary candidate pair. Otherwise t_i is solitary and $\mathit{level}(t_i)$ itself jumps. Clearly, $\mathit{level}(t_i) = \ell_j$ for some $j > i$. Let s''_i be a solitary task on level ℓ_i that is independent of s_j. Such a task exists because $\mathit{high}(s_j) \geq \mathit{high}(t_i) \geq \ell_i$. Then the pair (s''_i, s_j) is the secondary candidate pair for the jump from level ℓ_i.

For a candidate pair (t,t'), let $(t,t')_1$ denote t and let $(t,t')_2$ denote t'. For $i = 1,\dots,k$, define the *implied candidate pair* for the jump from level ℓ_i as

$$I(\ell_i) := \begin{cases} \mathcal{P}(\ell_i) & \text{if } \mathit{level}(t_i) \text{ does not jump or} \\ & \mathcal{P}(\ell_i)_2 \neq I(\mathit{level}(t_i))_1, \\ \mathcal{S}(\ell_i) & \text{otherwise.} \end{cases} \tag{5.7}$$

Lemma 5.37 *The set of implied candidate pairs* $\{I(\ell_i) \,|\, 1 \leq i \leq k\}$ *is a candidate set for* $(T, \prec)$.

Proof. For each jumping level ℓ_i, there is an implied candidate pair (t,t') such that t is on level ℓ_i and t' is on $\mathit{level}(t_i)$, where ℓ_i and t_i are defined as before. Hence, the landing set in the set of implied candidate pairs implies the same jump sequence as $\{t_1,\dots,t_k\}$, a jump sequence which is lexicographically greater than or equal to the LMJ sequence. It follows that properties 1 and 2 of Definition 5.35 hold. It remains to show that all implied candidate pairs are disjoint. Assume the contrary and let $\ell > \ell'$ be levels such that $I(\ell)$ and $I(\ell')$ are not disjoint. Clearly, $I(\ell)_1 \neq I(\ell')_1$ since there is only one implied candidate pair for every level that jumps. Moreover, $I(\ell)_1 \neq I(\ell')_2$ since $I(\ell)_1$ is on level ℓ but $I(\ell')_2$ is on a level below ℓ' which is below ℓ. Hence, there remain two cases we have to consider:

case 1: $I(\ell)_2 = I(\ell')_2$. In this case, ℓ and ℓ' jump to the same level ℓ''. The second components of the primary candidate pairs are built from the landing set $\{t_1,\dots,t_k\}$ and each task from this landing set is used exactly once as a second component in a primary candidate pair. Hence, $\mathcal{P}(\ell)_2 \neq \mathcal{P}(\ell')_2$. It follows that one of the implied candidate pairs for ℓ and ℓ' is a secondary candidate pair. We will consider the case $I(\ell) = \mathcal{S}(\ell)$. The other case, $I(\ell') = \mathcal{S}(\ell')$, is equivalent for reasons of symmetry.

Assume that $I(\ell) = \mathcal{S}(\ell)$. Then $\mathcal{P}(\ell)_2 = I(\ell'')_1$, since otherwise we would have chosen the primary candidate pair as the implied candidate pair for ℓ. It follows that $\mathcal{P}(\ell')_2 \neq I(\ell'')_1$ because all primary candidate pairs have different tasks in their second component, as noted before. As a consequence, the implied candidate pair for ℓ' must be its primary candidate pair. Since $\mathcal{S}(\ell)_2$ is the reserved solitary task on level ℓ'' and $\mathcal{P}(\ell')_2$ is different from this reserved solitary task due to the way the set of primary candidate pairs is constructed, it holds that $I(\ell)_2 = \mathcal{S}(\ell)_2 \neq \mathcal{P}(\ell')_2 = I(\ell')_2$. A contradiction to the assumption that $I(\ell)_2 = I(\ell')_2$. Hence, the case assumption is wrong and $I(\ell)_2 \neq I(\ell')_2$.

case 2: $I(\ell)_2 = I(\ell')_1$. In this case, level ℓ jumps to ℓ'. Clearly, the implied candidate pair for ℓ is not its primary candidate pair, since otherwise $\mathcal{P}(\ell)_2 \neq I(\ell')_1$ and therefore $I(\ell)_2 = \mathcal{P}(\ell)_2 \neq I(\ell')_1$, which contradicts the case assumption. Hence, we can savely assume that the implied candidate pair for ℓ is its secondary candidate pair and $\mathcal{P}(\ell)_2 = I(\ell')_1$. Since $\mathcal{S}(\ell)_2$ is the reserved solitary task on level ℓ' and $\mathcal{P}(\ell)_2$ is different from this reserved solitary task due to the way the set of primary candidate pairs is constructed, it follows that $I(\ell)_2 = \mathcal{S}(\ell)_2 \neq \mathcal{P}(\ell)_2 = I(\ell')_1$. A contradiction to the case assumption. Hence, the case assumption is wrong and $I(\ell)_2 \neq I(\ell')_1$.

It follows that $I(\ell)$ and $I(\ell')$ are disjoint and hence all pairs in the set $\{I(\ell_i) \mid 1 \leq i \leq k\}$ are disjoint. We conclude that $\{I(\ell_i) \mid 1 \leq i \leq k\}$ is a candidate set for $(T, \prec)$. □

To compute the set of implied candidate pairs in parallel, we construct an *implication graph* that reflects the dependencies between primary and secondary candidate pairs. The implication graph is a directed acyclic graph $H = (V, E)$ with vertex set V and edge set E. The vertex set consists of one vertex for each primary candidate pair and one vertex for each secondary candidate pair. For the sake of convenience, we use $\mathcal{P}(\ell)$ and $\mathcal{S}(\ell)$ to denote the vertices in V. An edge from vertex $\mathcal{X}(\ell)$ to vertex $\mathcal{Y}(\ell')$, with $\mathcal{X}, \mathcal{Y} \in \{\mathcal{P}, \mathcal{S}\}$, expresses that $\mathcal{X}(\ell)$ is the implied candidate pair for level ℓ iff $\mathcal{Y}(\ell')$ is the implied candidate pair for level ℓ'. If the choice of the implied candidate pair for some level ℓ does not depend on the choice made for the level that level ℓ jumps to, then we mark either $\mathcal{P}(\ell)$ or $\mathcal{S}(\ell)$, whichever of them is the implied candidate pair. In other words, for each level ℓ_i that jumps, we mark vertices and add edges to E according to the following rules derived from (5.7):

case 1: *level*(t_i) does not jump. Then the primary candidate pair for ℓ_i becomes the implied candidate pair. We add no edges to E but mark $\mathcal{P}(\ell_i)$.

case 2: $\mathcal{P}(\ell_i)_2 \neq \mathcal{P}(\mathit{level}(t_i))_1$ and $\mathcal{P}(\ell_i)_2 \neq \mathcal{S}(\mathit{level}(t_i))_1$. Then the primary candidate pair for ℓ_i is chosen as its implied candidate pair independent of the choice made for the implied candidate pair for *level*(t_i). We add no edges to E but mark $\mathcal{P}(\ell_i)$.

case 3: $\mathcal{P}(\ell_i)_2 = \mathcal{P}(\mathit{level}(t_i))_1$ and $\mathcal{P}(\ell_i)_2 = \mathcal{S}(\mathit{level}(t_i))_1$. Then the secondary candidate pair for ℓ_i is chosen as its implied candidate pair independent of the choice made for the implied candidate pair for *level*(t_i). In this case, we mark $\mathcal{S}(\ell_i)$.

case 4: $\mathcal{P}(\ell_i)_2 \neq \mathcal{P}(\mathit{level}(t_i))_1$ and $\mathcal{P}(\ell_i)_2 = \mathcal{S}(\mathit{level}(t_i))_1$. Then the implied candidate pair for ℓ_i will be its primary candidate pair if *level*(t_i) chooses its primary candidate pair, but will otherwise be its secondary candidate pair. In this case, we add $(\mathcal{P}(\ell_i), \mathcal{P}(\mathit{level}(t_i)))$ and $(\mathcal{S}(\ell_i), \mathcal{S}(\mathit{level}(t_i)))$ to E.

case 5: $\mathcal{P}(\ell_i)_2 = \mathcal{P}(\mathit{level}(t_i))_1$ and $\mathcal{P}(\ell_i)_2 \neq \mathcal{S}(\mathit{level}(t_i))_1$. Then the implied candidate pair for ℓ_i will be its primary candidate pair if *level*(t_i) chooses its secondary candidate pair, and it will be its secondary candidate pair if $level(t_i)$ chooses its primary candidate pair. Hence, we add $(\mathcal{P}(\ell_i), \mathcal{S}(level(t_i)))$ and $(\mathcal{S}(\ell_i), \mathcal{P}(level(t_i)))$ to E.

Note that the implication graph is a forest of intrees and every root is either marked or not. After a pointer jumping operation, each vertex in H knows its root vertex and can therefore determine whether its root is marked. A candidate pair is an implied candidate pair iff its root vertex is marked. Hence, a prefix operation suffices to obtain the set of implied candidate pairs.

An implication graph for the task system in Figure 5.1 is depicted in Figure 5.9. As before, we have chosen $\{t_{15}, t_{12}, t_{11}, t_9, t_4\}$ as reserved tasks and obtained the landing set $\{t_{10}, e_1, t_6, t_5, t_1\}$ from the matching in Figure 5.8. We select the solitary tasks t_{15}, t_{12}, t_{11}, t_6, t_5 to jump from the respective levels. Hence, (t_{15},t_{10}), (t_{12},e_1), (t_{11},t_6), (t_6,t_5), (t_5,t_1) are the primary candidate pairs. For levels 7, 5, and 2, the primary candidate pairs are used as secondary candidate pairs, since the fill-ins for these levels, as proposed by the landing set, are not solitary. For the jumps from level 4 and 3, the secondary candidate pairs differ from the primary candidate pairs. For these levels, the reserved solitary tasks on level 3, respectively 2, *i.e.*, tasks t_9 and t_4, are the alternative fill-ins. The only solitary task on level 4 that can be paired with task t_9 is task t_{11}, and for the jump from level 3 we choose task t_9. Hence, (t_{11},t_9) and (t_9,t_4) are the secondary candidate pairs for the jumps from level 4 respectively level 3. The set of implied candidate pairs obtained from the implication graph is $\{(t_{15},t_{10}), (t_{12},e_1), (t_{11},t_6), (t_9,t_4), (t_5,t_1)\}$, which happens to be the set of actual jumps used in the schedule in Figure 5.1.

We close this section by stating the main result of this chapter.

Theorem 5.38 *There exists a parallel algorithm that computes an optimal two processor schedule for any task system in time $O(\log^2 n)$ using $n^3/\log n$ processors of a CREW PRAM, where n is the number of tasks.*

Proof. Let us recall the various steps of our scheduling algorithm and their resource requirements. We first have to determine the level of each task and the total number of levels. As noted before, we assume that the precedence graph is given as an adjacency matrix. We compute an all-pairs longest path matrix B in $O(\log^2 n)$ time on $n^3/\log n$ processors (Theorem 3.26). The level of a task t relative to t' is $B(t,t')$. To determine the number of levels L, a prefix-maxima operation on B suffices, which can be performed in $O(\log n)$ time using $n^3/\log n$ processors.

To decide which levels jump, we have to augment the precedence graph by $2L+n+2$ tasks. Then, we apply the distance algorithm to the augmented

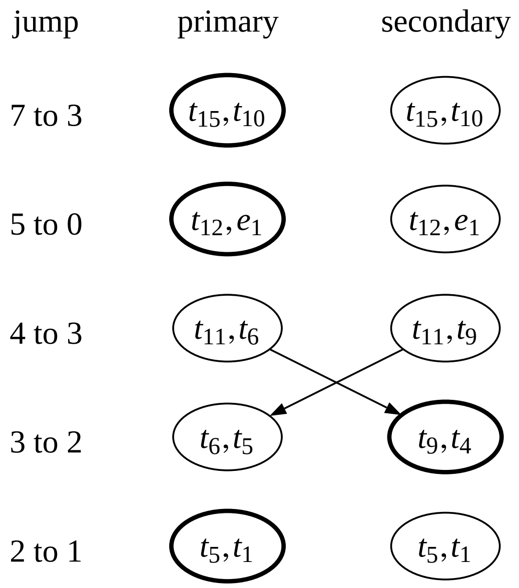


Figure 5.9: *An implication graph with primary and secondary candidate pairs and their dependencies. A candidate pair is implied iff the root of the intree it belongs to is marked (thick lined oval).*

precedence graph, which takes $O(\log^2 n)$ time on $n^3 / \log n$ processors (Theorem 5.28). It is now easy to determine the solitary tasks and the levels that jump (Lemma 5.4, Lemma 5.5, and Lemma 5.6). On each level that jumps, we reserve a solitary task with maximum high-value using a prefix-maxima operation segmented by levels. Then we construct a landing set that implies the LMJ sequence but does not contain any of the reserved solitary tasks. According to Lemma 5.33, this requires $O(\log^2 n)$ time using $n^2 / \log n$ processors. Next, we determine the set of implied candidate pairs as described above. Clearly, we can construct the primary candidate pairs, the secondary candidate pairs and then the implication graph using the given resources. The same holds for pointer jumping that we use to determine the roots of the implication graph.

According to Lemma 5.36 and Lemma 5.37, the set of implied candidate pairs obtained in the previous step forms the actual jumps of an LMJ schedule. The tasks not used in the implied candidate pairs are paired up within levels. Since there are no precedence constraints between tasks on the same level, this pairing can be done arbitrarily. An LMJ schedule is obtained by sorting the resulting set of task pairs, both for jumps and within levels. Clearly, this last step can be done within the desired resource bounds. □

5.9 Maximum Matchings in Co-Comparability Graphs

In this section we consider a problem closely related to the two processor scheduling problem: Maximum matchings in co-comparability graphs. The problem of computing maximum matchings in graphs is a fundamental problem in computer science. The first sequential algorithm for computing maximum matchings is due to Edmonds and requires time $O(|V|^4)$ [Edm65]. Since then a number of algorithms have been proposed. The fastest of them requires $O(\sqrt{|V|}|E|)$ time [MV80, Vaz94].

The parallel complexity of the general maximum matching problem is still open. There is no $\mathcal{NC}$ algorithm known, and no $\mathcal{P}$-completeness proof has been found. The only known fast parallel algorithms are randomized, *i.e.*, they belong to the class $\mathcal{RNC}$ [KUW86, Kar86, MVV87, GP88]. Although their *expected* running time is polylogarithmic, they may take an arbitrarily long time to halt. For some restricted classes of graphs, $\mathcal{NC}$ algorithms are known. This includes regular bipartite graphs [LPV81], convex bipartite graphs [DS84], interval graphs [MJ89], cographs [LO94], and co-comparability graphs [KVV85, HM86].

The class of co-comparability graphs is quite rich. It contains interval graphs and permutation graphs, and the latter class contains cographs. Co-comparability graphs belong to the class of *perfect graphs* [Gol80]. A graph G is called perfect if the maximum clique size and the chromatic number of every induced subgraph of G coincide. A graph G is called a *comparability graph* if the edges of G can be oriented such that the resulting directed graph is a transitively closed acyclic graph. More formally:

Definition 5.39 *Let $G = (V,E)$ be an undirected graph. Then G is a* comparability graph *if there exists a directed graph $\vec{G} = (V,\vec{E})$, called a* transitive orientation *for G, such that*

1. *for every $\{x,y\} \in E$, either $(x,y) \in \vec{E}$ or $(y,x) \in \vec{E}$,*
2. *if $(x,y) \in \vec{E}$ then $\{x,y\} \in E$, and*
3. *if $(x,y) \in \vec{E}$ and $(y,z) \in \vec{E}$ then $(x,z) \in \vec{E}$.*

Clearly, not every graph has a transitive orientation. A *co-comparability graph* is the complement of a comparability graph. The following theorem provides an alternative characterization of comparability graphs.

Theorem 5.40 ([GH64]) *A graph is a comparability graph iff each cycle of odd length has a triangular chord.*

The relationship between the problem of computing a maximum matching in a co-comparability graph and the two processor scheduling problem has first been recognized by Fujii, Kasami, and Ninomiya [FKN69]. They proposed an algorithm that constructs an optimal two processor schedule from a maximum matching in the incomparability graph of the given partial order. The proofs given in [FKN69] imply that the reverse is also possible:

Theorem 5.41 ([FKN69]) *Let G be a transitively closed precedence graph. Then the paired tasks in an optimal two processor schedule for G form a maximum matching in the undirected complement of G.*

Since a transitive orientation is a transitively closed precedence graph, all we need to compute maximum matchings in co-comparability graphs, is an algorithm for transitive orientations and an algorithm for the two processor scheduling problem. Given a co-comparability graph G, one can compute a maximum matching in G by first computing a transitive orientation for the complement of G and then computing an optimal two processor schedule S for the resulting precedence graph. By Theorem 5.41, the paired tasks in S form a maximum matching in G. The algorithms given in [KVV85, HM86] follow this line.

Recently, a new and efficient $\mathcal{NC}$ algorithm for computing transitive orientations has been proposed by Morvan and Viennot [MV96]. We combine this algorithm with the two processor scheduling algorithm presented in this chapter and obtain the following result.

Theorem 5.42 *Maximum matchings in co-comparability graphs can be computed on the CREW PRAM in time $O(\log^2 n)$ using n^3 processors.*

Proof. The transitive orientation algorithm in [MV96] runs on the CRCW PRAM and requires $O(\log n)$ time using n^3 processors. Hence, on the CREW PRAM, a transitive orientation can be computed in time $O(\log^2 n)$, because a CRCW PRAM can be simulated on an EREW PRAM with the help of sorting and a slowdown factor of $O(\log n)$. By Theorem 5.38, our two processor scheduling algorithm runs within the same resource bounds. □

Two well known subclasses of co-comparability graphs are interval graphs and permutation graphs. We obtain:

Corollary 5.43 *A maximum matching in an interval graph or a permutation graph can be computed on the CREW PRAM in time* $O(\log^2 n)$ *using* n^3 *processors.*

CHAPTER 6

Two Processor Schedules for Series Parallel Orders

In the last chapter we have proposed an algorithm that schedules arbitrary precedence constraints on two processors. Although this algorithm is more efficient than any previously known algorithm for the two processor scheduling problem, it is still not work optimal. One might ask whether a more restricted class of precedence constraints allows a more efficient algorithm. If precedence constraints are trees, then the two processor scheduling problem can be solved on the EREW PRAM by a time and work optimal algorithm, as we have shown in Chapter 4. A class of precedence constraints that includes trees is the class of series parallel orders. Series parallel orders occur quite often, in particular, they occur in parallel programs that follow the divide-and-conquer paradigm. In general, they are not "easier" to schedule than unrestricted precedence constraints. The m-processor UET scheduling problem is still $\mathcal{NP}$-complete if restricted to series parallel orders [May81, War81]. Moreover, no polynomial algorithm is known that computes optimal k-processor schedules for series parallel orders for any fixed k greater than two.

In this chapter we show that in the two processor case, series parallel orders can be scheduled much more efficiently than unrestricted precedence constraints. We present an algorithm that optimally schedules series parallel orders on two processors in time $O(\log n)$ using $n/\log n$ EREW PRAM processors, provided that a decomposition tree for the series parallel order is given. If the decomposition tree is not given, we combine our scheduling algorithm with an existing algorithm that decides whether a given graph is series parallel and, if that is the case, computes a binary decomposition tree. We obtain an algorithm that runs on the EREW PRAM in time $O(\log(n+e)+\log n \log^* n)$ and on the CRCW PRAM in time $O(\log(n+e))$, where e is the number of edges and n is the number of vertices in the given series parallel precedence graph. In either case, the number of required operations is $O(n+e)$. Note that this result implies a linear time sequential algorithm for the two processor scheduling problem with series parallel orders.

This chapter is organized as follows. In the next section, we recall some definitions related to series parallel orders and their decomposition trees. In Section 6.2, we give an overview of the strategy that we use to compute optimal two processor schedules for series parallel orders. In the section thereafter, we introduce “backbone tasks” and present some of their properties. In Section 6.4, a well formed word of parentheses is constructed from the decomposition tree that is used in the following section to modify the levels of tasks. In Section 6.6, we show that scheduling the tasks in nonincreasing order of modified levels yields an optimal two processor schedule. In the last section, we summarize the algorithm and show that it can be implemented to run efficiently in parallel.

6.1 Series Parallel Orders and Decomposition Trees

Let $(T,\prec)$ be a series parallel order, and let G be a series parallel precedence graph for $(T,\prec)$ (*i.e.*, G is a VSP dag, cf. Subsection 2.2.4). We assume in the sequel that G is given as a decomposition tree $B=(V,E)$. Recall that each leaf of B corresponds to a vertex in G and vice versa. Each internal vertex v of B is either marked with “P” or “S”, representing a series or parallel composition of the VSP dags that correspond to the subtrees of B rooted at the left child and the right child of v. We adopt the convention that the tasks in the left subtree of an “S” vertex in B precede the tasks in the right subtree.

Figure 6.1a shows a series parallel precedence graph consisting of sixteen tasks. In Figure 6.1b, a decomposition tree for it is depicted. An optimal two processor schedule for the task system given in Figure 6.1a is shown in Figure 6.2.

Recall that $root(B)$ denotes the root vertex of B, and $G(v)$ denotes the VSP dag that is defined by the subtree of B rooted at vertex v. Let $l(v)$ and $r(v)$ denote the left child, respectively right child, of vertex v, and let $type(v)$ be either “P” or “S”, indicating the kind of composition vertex v represents. Let $|v|$ denote the number of leaves in the subtree of B rooted at vertex v, *i.e.*, $|v|$

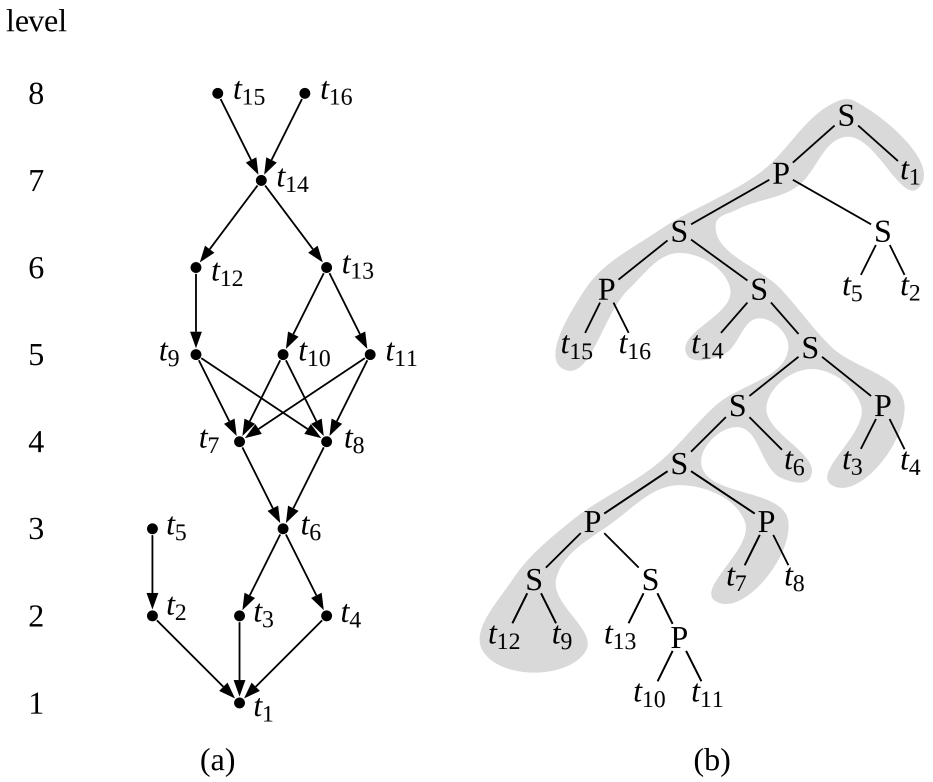


Figure 6.1: *(a) A series parallel precedence graph and (b) a decomposition tree for it.*

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

Figure 6.2: *An optimal two processor schedule for the given series parallel order.*

is the number of tasks in $G(v)$. For the sake of notational convenience, we identify the leaves in B with the tasks in G.

6.2 A Scheduling Strategy

In Section 4.2 we have introduced the *highest level first* scheduling strategy. In this strategy, tasks are scheduled greedily level by level, starting with tasks on the highest level. Whenever there is a choice, choose the highest available task. This strategy is optimal for precedence graphs that are either inforests or outforests (Theorem 4.2). In the two processor case, HLF schedules are also optimal for series parallel orders. In fact, every two processor HLF schedule for $(T, \prec)$ is an LMJ schedule (cf. Section 5.1). To see this, consider the following. Assume that a certain subset of tasks has already been mapped to timesteps. Let H be the set of unscheduled tasks on level ℓ. Since $(T, \prec)$ is a series parallel order, we can derive that if a task in H can jump to some level ℓ', then all tasks in H can jump to level ℓ'. This has two consequences. First, whenever a level ℓ jumps, then we can use an arbitrary task to jump from ℓ, since all unscheduled tasks on level ℓ can jump equally high. Second, whenever the highest level ℓ' we can jump to contains more than one task that can be used as a fill-in, then we can choose any of these tasks as a fill-in task, since the tasks that remain unscheduled on level ℓ' can jump equally high. As a consequence, the jump sequence of an HLF schedule is an LMJ sequence. Hence, every two processor HLF schedule for a series parallel order is an LMJ schedule and therefore optimal (Theorem 5.1).

Our parallel algorithm, however, does not compute HLF schedules. Instead, we use a scheduling strategy that enables us to exploit the structure of the given decomposition tree. In the following we give a rough sketch of our strategy. For an illustration, we refer to Figure 6.3. In a first step, we select an arbitrary longest path in the precedence graph. The tasks on this path are called *backbone tasks*. All other tasks are called *free*. In Figure 6.3a, the tasks on the leftmost path from the source to the sink are the backbone tasks. Our goal is to find for each of these backbone tasks a free task that it can be paired with in a schedule. We concatenate free tasks and thread them on a number of strings like pearls. The order in which free tasks are threaded is not important as long as predecessors come before their successors. For reasons of efficiency, we closely follow the structure of the given decomposition tree for threading free tasks. In Figure 6.3a, the strings of tasks are indicated by grey curves. Then, we pull these strings one after the other (in a specific order) towards higher levels. We try to “move” as many free tasks as possible but to each level at most one free task is “moved”. The level a free task is moved to is called its *modified level*. All tasks not moved retain their old level. In Figure 6.3b, the strings have already been pulled and some of the free tasks have been moved to higher levels. In the end, we schedule tasks in the order of nonincreasing modified levels.

This strategy does not result in HLF schedules. However, if tasks are threaded carefully, it produces optimal two processor schedules for series parallel orders. As already mentioned, task threading closely follows the structure of the decomposition tree. Pulling the strings is done by computing the matching pairs in a well formed word of parentheses. Each backbone task corresponds to an opening parenthesis in this word, and each free task corresponds to a closing parenthesis. Details are given in the following sections. We start by introducing backbone tasks.

6.3 Backbone Tasks

We assume in the sequel that for all “P” vertices v of the decomposition tree, $G(l(v))$ is at least as high as $G(r(v))$. If this is not the case, we swap the left and right child of v. Let $\beta(B)$ denote the set of all vertices of B that have no ancestor that is the right child of a “P” vertex. The leaves in $\beta(B)$ form a longest path in G. We call this longest path *backbone*, and the tasks on it are called *backbone tasks*. In Figure 6.1b, the vertices that belong to $\beta(B)$ are shaded. The backbone tasks are t_1, t_3, t_6, t_7, t_9, t_{12}, t_{14}, and t_{15}. In the following, we explore the relationship between backbone tasks and the preorder numbers of tasks in the decomposition tree.

Lemma 6.1 *Let x be the task with smallest preorder number in B among tasks on the level of x. Then x is a backbone task.*

Proof. Assume that x is not on the backbone. Since on every level there exists exactly one backbone task, there must be a backbone task y on the level of x. The preorder number of y in B is greater than that of x. It follows that x and y have a common ancestor v such that x is in the left subtree of v and y is in the right subtree of v. If v represents a parallel composition, then y has an ancestor that is the right child of a “P” vertex. Hence, y is not a backbone task. A contradiction. Otherwise, v represents a series composition. But then, x is a predecessor of y. Again, a contradiction. Hence, our assumption is wrong and x is on the backbone. □

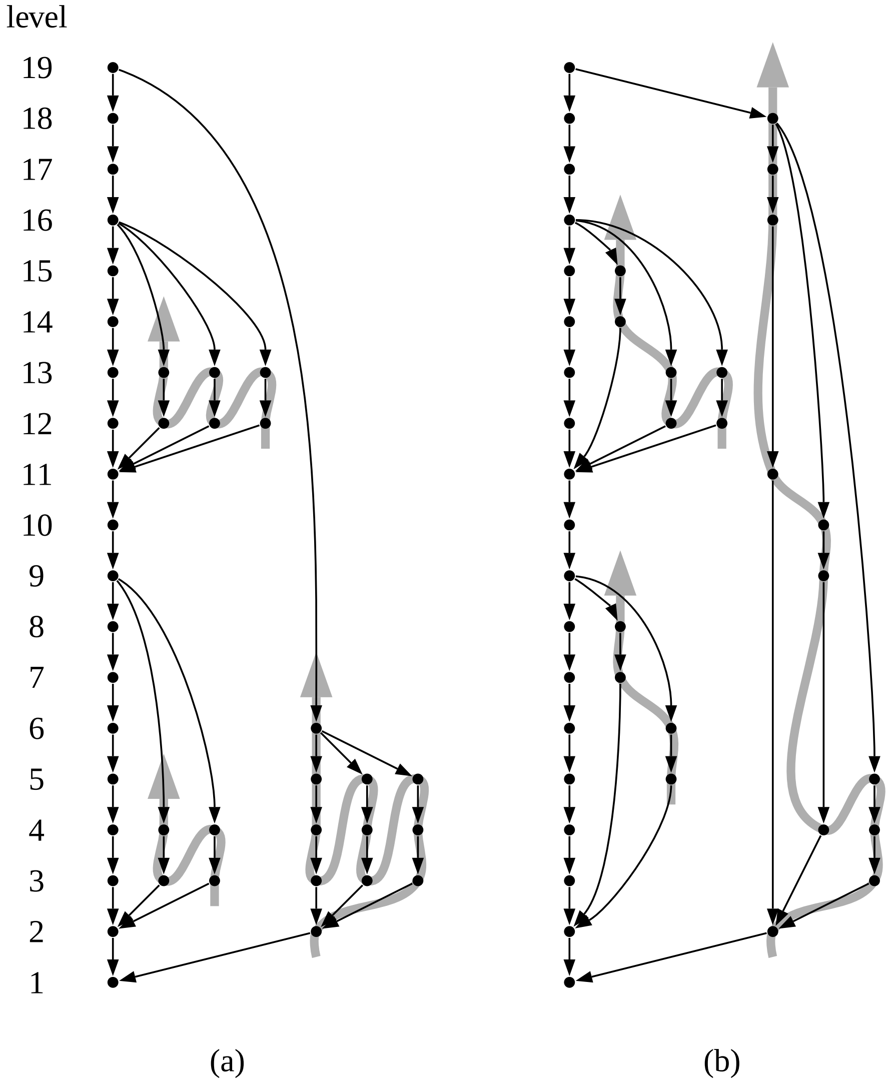


Figure 6.3: *A series parallel precedence graph. (a) Free tasks are threaded on strings like pearls. (b) We "pull" the strings and move some of the free tasks to higher levels. (See text.)*

Lemma 6.2 *Let y be a backbone task, and let x be a predecessor of y. Then every task with a preorder number in B smaller than that of x is also a predecessor of y.*

Proof. Let p_y be the path from y to the root of B. Since x is a predecessor of y, there exists an "S" vertex v in p_y such that x is in the left subtree of v and y is in the right subtree of v. Consider the path p_x from x to the root of B, and let z be a task with a preorder number in B smaller than that of x. Then there exists a vertex w on the path p_x such that z is in the left subtree of w and x is in the right subtree of w. If w represents a series composition, then z is a predecessor of x, and hence a predecessor of y (Figure 6.4a and 6.4b). Otherwise, w represents a parallel composition. Assume that w is part of p_y (Figure 6.4d). Then y would have an ancestor that is the right child of a "P" vertex. But this contradicts the fact that y is on the backbone. Hence, w is in the left subtree of v (Figure 6.4c), and therefore, z is a predecessor of y. □

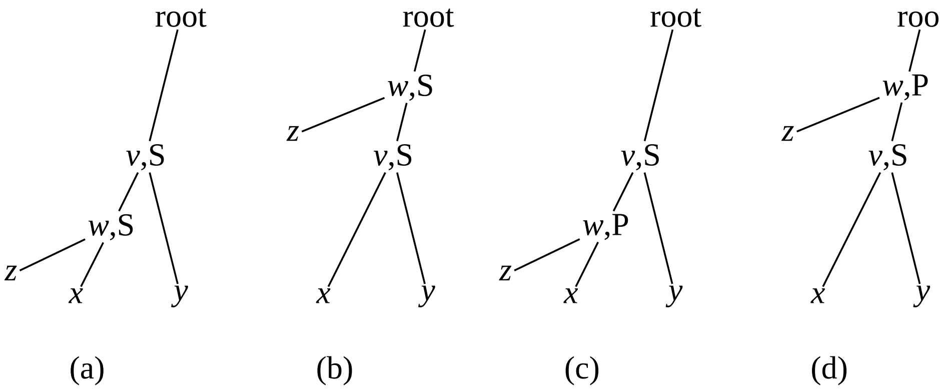


Figure 6.4: *Task x is a predecessor of y and the preorder number of z in the decomposition tree is smaller than that of x. Hence, one of the cases (a)–(d) applies in the decomposition tree. In case (d), y is not a backbone task. (See proof of Lemma 6.2.)*

6.4 Matching Parentheses in the Decomposition Tree

To construct an optimal schedule, it is helpful to know, for every vertex v of the decomposition tree, the length of an optimal two processor schedule for the task system that corresponds to the precedence graph $G(v)$. In the following definition, we give an "obvious" lower bound on this scheduling length.

Definition 6.3 *Let v be a vertex in a decomposition tree. We define recursively on the structure of the decomposition tree*

$$\omega(v) := \begin{cases} 1 & \textit{if } v \textit{ is a leaf}, \\ \omega(l(v)) + \omega(r(v)) & \textit{if type}(v) = \text{“S”}, \\ \max(\omega(l(v)), \lceil |v|/2 \rceil) & \textit{if type}(v) = \text{“P”}. \end{cases}$$

Clearly, any two processor schedule for the tasks of $G(v)$ has length at least $\omega(v)$. As it will turn out later, $\omega(v)$ is actually the optimal length.

The main idea in our parallel algorithm is to use the matching pairs in a well formed word of parentheses to find tasks that can be paired with backbone tasks. We wish to match each backbone task x with a task y from some lower level such that x and y can be mapped to the same timestep. To this end, we first assign parentheses to the vertices in the decomposition tree.

Definition 6.4 *Let v be a vertex in a decomposition tree* $B = (V, E)$. *The* parenthesis label *of v, denoted by* $\lambda(v)$*, is defined as follows:*

$$\lambda(v) := \begin{cases} \text{“(”} & \textit{if } v \textit{ is a leaf in } \beta(B), \\ \text{“)”} & \textit{if } v \textit{ is a leaf in } V - \beta(B), \\ \text{“}\overbrace{((\ldots(}^{k}\text{”} & \textit{if type}(v) = \text{“P” } \textit{and } v \in \beta(B), \\ \varepsilon & \textit{otherwise}, \end{cases}$$

where $k = \max(0, 2\lceil |v|/2 \rceil - 2\omega(l(v)))$.

The parenthesis labels in the decomposition tree are concatenated to form a parenthesis word.

Definition 6.5 *Let v be a vertex in a decomposition tree B. The* parenthesis word *of v, denoted by* $\pi(v)$*, is recursively defined as follows:*

$$\pi(v) := \begin{cases} \lambda(v) & \textit{if } v \textit{ is a leaf}, \\ \lambda(v)\,\pi(l(v))\,\pi(r(v)) & \textit{if type}(v) = \text{“P” } \textit{and } v \in \beta(B), \\ \pi(r(v))\,\pi(l(v)) & \textit{if type}(v) = \text{“S” } \textit{and } v \in \beta(B), \\ \pi(l(v))\,\pi(r(v)) & \textit{otherwise}. \end{cases}$$

The parenthesis word of v is not a well formed word of parentheses. If v is part of $\beta(B)$, then all closing parentheses in $\pi(v)$ are matched, but a number of opening parentheses are unmatched. This number corresponds to the number of partial timesteps in an optimal schedule for the tasks in $G(v)$.

Lemma 6.6 *Let B be a decomposition tree, and let $v \in \beta(B)$. Then $\pi(v)$ followed by $2\omega(v) - |v|$ closing parentheses is a well formed word of parentheses.*

Proof. By induction on the structure of the decomposition tree. If v is a leaf, then $\pi(v) =$ "(" and the number of appended closing parentheses is 1.

Let v represent a series composition. Both the left and right child of v are contained in $\beta(B)$, since $v \in \beta(B)$. By inductive hypothesis, $\pi(l(v))$ followed by $2\omega(l(v)) - |l(v)|$ closing parentheses is a well formed word of parentheses, and the same holds for $\pi(r(v))$ followed by $2\omega(r(v)) - |r(v)|$ closing parentheses. Hence, $\pi(v) = \pi(r(v))\,\pi(l(v))$ followed by $2\omega(v) - |v| = 2\omega(r(v)) - |r(v)| + 2\omega(l(v)) - |l(v)|$ closing parentheses is well formed, too.

Now, let v represent a parallel composition. Since v is in $\beta(B)$, its left child is in $\beta(B)$, too, but its right child is not. By inductive hypothesis, $\pi(l(v))$ followed by $2\omega(l(v)) - |l(v)|$ closing parentheses is a well formed word of parentheses. The word $\pi(v)$ consists of $\max(0, 2\lceil |v|/2 \rceil - 2\omega(l(v)))$ opening parentheses, followed by $\pi(l(v))$, followed by $|r(v)|$ closing parentheses. We have to consider two cases.

case 1: $\omega(v) = \omega(l(v))$. Then $|v| \leq 2\omega(l(v))$. Therefore $\max(0, 2\lceil |v|/2 \rceil - 2\omega(l(v)))$ is zero. There are $2\omega(l(v)) - |l(v)|$ unmatched opening parentheses in $\pi(l(v))$. In $\pi(v)$, we append $|r(v)|$ closing parentheses. Hence, $2\omega(l(v)) - |l(v)| - |r(v)| = 2\omega(v) - |v|$ opening parentheses remain unmatched in $\pi(v)$.

case 2: Otherwise $\omega(v) = \lceil |v|/2 \rceil$ and $|v| > 2\omega(l(v))$, according to Definition 6.3. The number of unmatched opening parentheses in $\pi(v)$ is $2\lceil |v|/2 \rceil - 2\omega(l(v)) + 2\omega(l(v)) - |l(v)| - |r(v)| = 2\lceil |v|/2 \rceil - |v| = 2\omega(v) - |v|$.

We conclude that if we append $2\omega(v) - |v|$ closing parentheses to $\pi(v)$, then we obtain a well formed word of parentheses. □

Lemma 6.7 *Let v be a* "P" *vertex in $\beta(B)$ such that $\lambda(v) \neq \varepsilon$. Then at most one opening parenthesis in $\pi(v)$ is not matched by a closing parenthesis in $\pi(v)$, and if there is such an unmatched opening parenthesis, then it belongs to $\lambda(v)$.*

Proof. Since $\lambda(v) \neq \varepsilon$, it holds that $\max(0, 2\lceil |v|/2 \rceil - 2\omega(l(v))) > 0$. It follows that $\omega(v) = \lceil |v|/2 \rceil$, by Definition 6.3. By Lemma 6.6, all closing parentheses of $\pi(v)$ are matched within $\pi(v)$ and the number of unmatched opening parentheses in $\pi(v)$ is $2\omega(v) - |v|$. Since $\omega(v) = \lceil |v|/2 \rceil$, this number is either zero or one, depending on whether $|v|$ is even or odd.

We observe that $\lambda(v)$ consists of opening parentheses only and in $\pi(l(v))$ all closing parentheses are matched within $\pi(l(v))$ (Lemma 6.6). Furthermore, $\pi(r(v))$ consists of closing parentheses only, because the right subtree of v does not belong to $\beta(B)$. Since $\pi(v) = \lambda(v)\,\pi(l(v))\,\pi(r(v))$, we can derive that if there is an unmatched opening parenthesis in $\pi(v)$, then this parenthesis is the first parenthesis of $\lambda(v)$. □

As it turns out, it is helpful to introduce empty tasks. For each partial timestep, we imagine an empty task being mapped to it, indicating that one of the two processors has to stay idle during this timestep. We add $2\omega(root(B)) - |root(B)|$ empty tasks $e_1, e_2, \ldots$ to the given task system. Since $\omega(root(B))$ is the length of an optimal schedule for $(T, \prec)$ (for technical reasons, the proof of this fact is deferred), every optimal schedule for the extended task system has no partial timesteps anymore. Moreover, every optimal schedule for the extended task system is also an optimal schedule for the original task system. A decomposition tree for the extended task system can be obtained from B as follows. We build a proper binary tree with $2\omega(root(B)) - |root(B)| + 1$ leaves, where all internal vertices represent parallel compositions. We replace the leftmost leaf by B and let all other leaves correspond to empty tasks. We call the resulting decomposition tree *extended*, and denote it by $\hat{B}$. Note that the parenthesis word of $root(\hat{B})$ is a well formed word of parentheses, since $2\omega(root(\hat{B})) - |root(\hat{B})|$ is zero.

In Figure 6.5, an extended decomposition tree $\hat{B}$ for the task system given in Figure 6.1a is shown. At each vertex we have drawn its parenthesis label. This extended decomposition tree $\hat{B}$ has been obtained from the decomposition tree in Figure 6.1b by adding two empty tasks e_1 and e_2 using parallel compositions. The vertices that belong to $\beta(\hat{B})$ are shaded. Note that only one nonleaf vertex of the tree has a nonempty parenthesis label, namely, the "P" vertex that is the lowest common ancestor of t_{12} and t_{13}.

Definition 6.8 *Let $\hat{B} = (\hat{V}, \hat{E})$ be an extended decomposition tree, and let $v \in \hat{V} - \beta(\hat{B})$. Then the* match *of v, denoted by $m(v)$, is the vertex in $\hat{B}$ that belongs to the opening parenthesis matching the closing parenthesis in $\pi(root(\hat{B}))$ that v belongs to.*

Note that $m(v)$ is well defined, since $\pi(root(\hat{B}))$ is a well formed word of parentheses. If $w = m(v)$, then we also say that v and w are a *matching pair* or *v matches w* and w matches v in $\hat{B}$.

6.5 Modified Levels

Guided by the matched pairs in the parenthesis word of the extended decomposition tree, we adjust the levels of tasks. The new levels are called *modified*

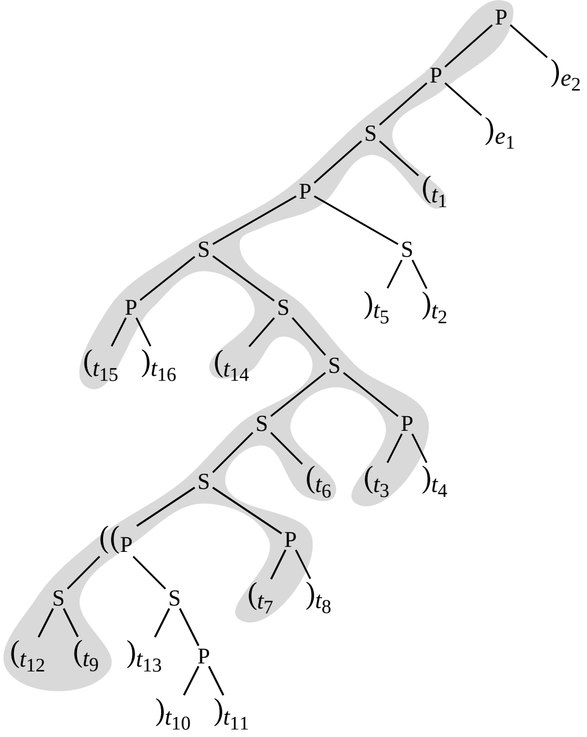


Figure 6.5: *The extended decomposition tree $\hat{B}$ for the given task system and its parenthesis labels.*

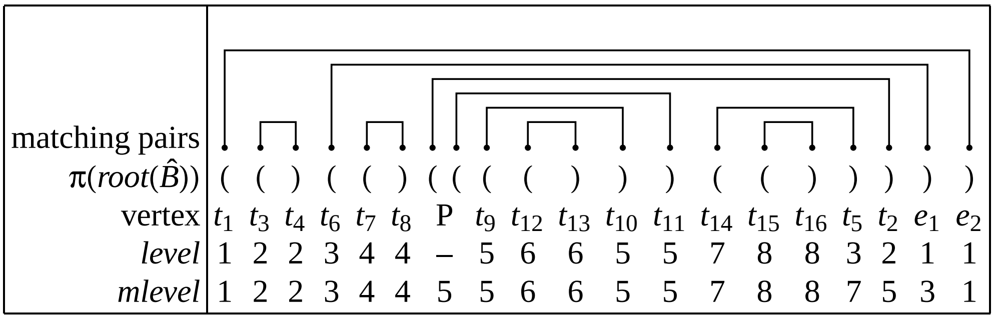
levels. Modified levels are assigned to each vertex in the decomposition tree with a nonempty parenthesis label. Hence, not only tasks have modified levels but also “P” vertices in $\beta(\hat{B})$. We first assign modified levels to vertices in $\beta(\hat{B})$. The modified level of a backbone task is its original level. The modified level of a “P” vertex v in $\beta(\hat{B})$ is the lowest level of a task contained in the subtree rooted at v. Then, modified levels are assigned to all remaining tasks. The modified level of a task x not in $\beta(\hat{B})$ is the maximum of its original level and the modified level of the vertex/task that matches x in $\hat{B}$. All other vertices of the decomposition tree, *i.e.*, all “S” vertices, have no modified level.

Definition 6.9 *Let $\hat{B} = (\hat{V}, \hat{E})$ be the extended decomposition tree, and let $v \in \hat{V}$ with $\lambda(v) \neq \varepsilon$. The* modified level *of v, denoted by $mlevel(v)$, is defined as* *follows:*

$$mlevel(v) := \begin{cases} level(v) & \textit{if } v \textit{ is a leaf in } \beta(\hat{B}), \\ \max(mlevel(m(v)), level(v)) & \textit{if } v \textit{ is a leaf in } \hat{V} - \beta(\hat{B}), \\ \min_{u \in G(v)}\{level(u)\} & \textit{if } type(v) = \text{“P”} \textit{ and } v \in \beta(\hat{B}). \end{cases}$$

We say x is a *moved task* in $\hat{B}$, or x is *moved* in $\hat{B}$, if its modified level differs from its original level. Clearly, if x is moved, then $mlevel(x) > level(x)$ and x is not a backbone task.

In Figure 6.6, the parenthesis word of the root vertex of the extended decomposition tree for our example is given (cf. Figure 6.5). For each vertex with a nonempty parenthesis label, its label, its level, and its modified level is depicted. Furthermore, the matching pairs are indicated. The moved tasks are t_5, t_2, e_1, and e_2, because the tasks they are matched with are on modified levels that are higher than their own levels.



matching pairs																			
$\pi(root(\hat{B}))$	(	(	)	(	(	)	((	(	(	)	)	)	(	(	)	)	)	)	)
vertex	t_1	t_3	t_4	t_6	t_7	t_8	P	t_9	t_{12}	t_{13}	t_{10}	t_{11}	t_{14}	t_{15}	t_{16}	t_5	t_2	e_1	e_2
level	1	2	2	3	4	4	–	5	6	6	5	5	7	8	8	3	2	1	1
mlevel	1	2	2	3	4	4	5	5	6	6	5	5	7	8	8	7	5	3	1

Figure 6.6: *The parenthesis word of the root of the extended decomposition tree $\hat{B}$ with matching pairs, task levels, and modified levels.*

In the following we investigate some properties of modified levels.

Lemma 6.10 *Let x and y be moved tasks in the extended decomposition tree $\hat{B}$. Then $mlevel(x) \neq mlevel(y)$.*

Proof. The parenthesis label of both x and y is a closing parenthesis. Let v be the vertex that matches x, and let w be the vertex that matches y. Clearly, $v \in \beta(\hat{B})$ and $w \in \beta(\hat{B})$. If v and w are both leaves in B (*i.e.*, tasks), then their modified level is still their original level. Their levels are not identical, since they belong to the same longest path. As a consequence, the modified levels of x and y are different, too. Otherwise, at least one of v and w is not a leaf but a “P” vertex. Without loss of generality, let v be a “P” vertex. Then w is either a leaf or a “P” vertex as well. We have to consider two cases.

case 1: v and w are not on the same path in $\hat{B}$. Let u be the lowest common ancestor of v and w. Clearly, u represents a series composition, because otherwise one of v and w would not be part of $\beta(\hat{B})$. Assume first that v is in the left subtree of u and w is in the right subtree of u. It follows that all tasks in $G(v)$ are predecessors of all tasks in $G(w)$. Hence, the lowest level of a task in $G(v)$ is higher than the lowest level of a task in $G(w)$, *i.e.*, $\min_{x \in G(v)}\{\mathit{level}(x)\} > \min_{x \in G(w)}\{\mathit{level}(x)\}$. Conversely, if w is in the left subtree of u and v is in the right subtree, then $\min_{x \in G(w)}\{\mathit{level}(x)\} > \min_{x \in G(v)}\{\mathit{level}(x)\}$. If w is a leaf, then replace $G(w)$ by w and replace $\min_{x \in G(w)}\{\mathit{level}(x)\}$ by $\mathit{level}(w)$. In any case, the modified levels of v and w are not identical, and as a consequence, the modified levels of x and y are different, too.

case 2: v and w are on the same path in $\hat{B}$. Without loss of generality, we can assume that v is an ancestor of w. (If w is a leaf, then it is clear that v is an ancestor of w. Otherwise, both v and w are “P” vertices and it does not matter which of them is the ancestor.) Since the parenthesis label of v is not empty, the only opening parenthesis in $\pi(v)$ not matched within $\pi(v)$ belongs to $\lambda(v)$, by Lemma 6.7. It follows that the opening parenthesis of w that is matched with the closing parenthesis of y is matched within $\pi(v)$. Hence, y is a descendant of v. Assume the modified level of v and that of w are identical. Then their modified level is the lowest level of a task in $G(v)$. The level of y is at least as high as the lowest level in $G(v)$. Hence, y is not a moved task. A contradiction. Otherwise, the modified levels of v and w are not identical. But then the same trivially holds for x and y. □

Lemma 6.11 *Let v and w be vertices in the extended decomposition tree $\hat{B}$. Let $\lambda(v)$ and $\lambda(w)$ consist of opening parentheses, and let $\lambda(v)$ appear before $\lambda(w)$ in $\pi(\mathit{root}(\hat{B}))$. Then $\mathit{mlevel}(v) < \mathit{mlevel}(w)$ or v is an ancestor of w.*

Proof. Vertices v and w are either backbone tasks or “P” vertices in $\beta(\hat{B})$. Let us first recall that the modified level of a backbone task is its original level and the modified level of a “P” vertex in $\beta(\hat{B})$ is the lowest level of a task in the subtree rooted at that vertex. Let u be the lowest common ancestor of v and w in $\hat{B}$. Clearly, $u \in \beta(\hat{B})$. We have to consider two cases.

case 1: $\mathit{type}(u) =$ “S”. Then $\pi(u) = \pi(r(u))\,\pi(l(u))$. It follows that v is part of the right subtree of u and w is part of the left subtree. All tasks in $G(r(u))$ are successors of all tasks in $G(l(u))$. Hence, all levels of tasks in $G(r(u))$ are lower than the levels in $G(l(u))$. Therefore, $\mathit{mlevel}(v) < \mathit{mlevel}(w)$.

case 2: $\mathit{type}(u) =$ "P". Then $\pi(u) = \lambda(u)\pi(l(u))\pi(r(u))$. We observe that neither v nor w belongs to the right subtree of u, since both are part of $\beta(\hat{B})$. It follows that $v = u$ and w belongs to the left subtree of u. In other words, v is an ancestor of w. □

Lemma 6.12 *Let $\hat{B}$ be the extended decomposition tree, and let x be a predecessor of y. Then $\mathit{mlevel}(x) > \mathit{mlevel}(y)$.*

Proof. Clearly, $\mathit{level}(x) > \mathit{level}(y)$. If the modified level of y is still its original level, then the lemma trivially holds, because the modified level of a task is not lower than its original level. Otherwise, y is a moved task and hence not on the backbone. There are two cases we have to consider.

case 1: x and y have a common ancestor that is the right child of a "P" vertex v in $\beta(\hat{B})$. Since $x \prec y$, the lowest common ancestor of x and y is an "S" vertex. Let u denote this lowest common ancestor of x and y. Clearly, u is not part of $\beta(\hat{B})$. Hence, $\pi(u) = \pi(l(u))\pi(r(u))$ and x belongs to the left subtree of u and y belongs to the right subtree. It follows that in $\pi(\mathit{root}(\hat{B}))$, the closing parenthesis of x appears before the closing parenthesis of y. Furthermore, since x and y have a common ancestor that is the right child of a "P" vertex, there is no opening parenthesis in $\pi(\mathit{root}(\hat{B}))$ between the closing parenthesis of x and that of y. It follows that in $\pi(\mathit{root}(\hat{B}))$, the opening parenthesis that matches the closing parenthesis of x is behind the opening parenthesis that matches y. By Lemma 6.11, $\mathit{mlevel}(m(x)) > \mathit{mlevel}(m(y))$ or $m(y)$ is an ancestor of $m(x)$. In the former case, $\mathit{mlevel}(x) > \mathit{mlevel}(y)$, and we are done.

Hence, assume that the latter case holds, *i.e.*, $m(y)$ is an ancestor of $m(x)$. It follows that $m(y)$ is a "P" vertex. By Lemma 6.6, all closing parentheses in $\pi(v)$ are matched within $\pi(v)$. Hence $m(y)$ belongs to the subtree of $\hat{B}$ rooted at v. Assume $m(y) = v$. The modified level of v is the lowest level of a task in $G(v)$. Hence, $\mathit{mlevel}(v) \leq \mathit{level}(y)$. As a consequence, y is not a moved task, a contradiction. Otherwise, $m(y)$ is part of the left subtree of v. By Lemma 6.7, the only opening parenthesis unmatched in $\pi(m(y))$ is part of $\lambda(m(y))$. As a consequence, the opening parentheses in the parenthesis label of $m(x)$ are matched within $\pi(m(y))$, since $m(x)$ is a descendant of $m(y)$. We obtain a contradiction, because one of the opening parenthesis of $\lambda(m(x))$ and the closing parenthesis of x are supposed to be a matched pair. We conclude that our assumption that $m(y)$ is an ancestor of $m(x)$ is wrong. Therefore $\mathit{mlevel}(m(x)) > \mathit{mlevel}(m(y))$ and hence $\mathit{mlevel}(x) > \mathit{mlevel}(y)$.

case 2: x and y do not have a common ancestor that is the right child of a "P" vertex in $\beta(\hat{B})$. Let w be the "P" vertex in $\beta(\hat{B})$ such that y is part of the right subtree of w. Such a vertex exists, because y is not on the backbone.

Since $x \prec y$, the lowest common ancestor u of x and y is an "S" vertex. Vertex u is not part of the right subtree of w, since otherwise case 1 applies. Hence, u is also the lowest common ancestor of x and w and x belongs to the left subtree of u and w belongs to the right subtree of u. It follows that the level of x is higher than the levels of tasks in $G(w)$. The modified level of x is not lower than its original level and the modified level of any vertex in $G(w)$ with a parenthesis label consisting of opening parentheses is not higher than the highest level of a task in $G(w)$. By Lemma 6.6, all closing parentheses in $\pi(w)$ are matched within $\pi(w)$. Hence, $mlevel(x) > mlevel(m(y))$. It follows that $mlevel(x) > mlevel(y)$. □

Lemma 6.13 *Let x be a backbone task in the extended decomposition tree $\hat{B}$ such that for every task y with $mlevel(y) > mlevel(x)$ it holds that either $y \prec x$ or y is a moved task. Then the number of tasks on modified levels higher than that of x is even.*

Proof. Let F denote the set of all tasks on modified levels higher than that of x. The set F consists of all predecessors of x plus all moved tasks. Note that $mlevel(x) = level(x)$ since x is a backbone task. Let $y \in F$. Then there exists a distinct vertex v in $\hat{B}$ that is matched with y. There are some cases we have to consider.

case 1: y is a backbone task. Then its parenthesis label is an opening parenthesis, and v is a task with a closing parenthesis. It follows that $mlevel(v) \geq mlevel(y)$. Since $mlevel(y) > mlevel(x)$, we obtain that $mlevel(v) > mlevel(x)$. It follows that $v \in F$.

case 2: y is not a backbone task. Then its parenthesis label is a closing parenthesis and v is either a backbone task or a "P" vertex in $\beta(\hat{B})$.

case 2.1: v is a backbone task. There are two subcases.

case 2.1.1: y is a moved task. Then $mlevel(y) = mlevel(v)$. Since $mlevel(y) > mlevel(x)$, we obtain that $mlevel(v) > mlevel(x)$. It follows that $v \in F$.

case 2.1.2: $y \prec x$. Then the lowest common ancester u of y and x is an "S" vertex, and x is in the right subtree of u while y is in the left subtree of u. Since x is on the backbone, u is part of $\beta(\hat{B})$. Since y is not on the backbone, there must exist a "P" vertex $w \in \beta(\hat{B})$ in the left subtree of u such that y is in the right subtree of w. By Lemma 6.6, all closing parentheses of $\pi(w)$ are matched within $\pi(w)$. Hence, the match of y, namely v, is contained in the subtree of $\hat{B}$ rooted at w. As a consequence, $v \prec x$. Since both tasks are on the backbone, we obtain $mlevel(v) > mlevel(x)$. Hence, $v \in F$.

case 2.2: v is a "P" vertex in $\beta(\hat{B})$. By definition, the parenthesis label of v consists of an even number of opening parentheses. We claim that all tasks matched with these parentheses are contained in F. Clearly, one of these tasks is y. Again, there are two subcases we have to consider.

case 2.2.1: y is a moved task. Then $mlevel(y) = mlevel(v)$. For every task z that is matched with an opening parenthesis in $\lambda(v)$, it holds that $mlevel(z) \geq mlevel(v)$. Since $mlevel(y) > mlevel(x)$, we obtain that $mlevel(z) > mlevel(x)$. It follows that $z \in F$.

case 2.2.2: $y \prec x$. We can apply similar arguments as in case 2.1.2. Let u and w be as in case 2.1.2. Then v is part of the subtree of $\hat{B}$ rooted at w (possibly $w = v$). All tasks in $G(w)$ are predecessors of x and hence they are on higher levels than x. As a consequence, the modified level of v is higher than the level of x. For every task z matched with an opening parenthesis in $\lambda(v)$, it holds that $mlevel(z) \geq mlevel(v)$. Since $mlevel(v) > mlevel(x)$, we obtain that $mlevel(z) > mlevel(x)$. It follows that $z \in F$.

We have shown that every task $y \in F$ either is matched with a distinct task in F or it belongs to a set of tasks matched with opening parentheses of a “P” vertex and this set is part of F and has an even size. Consequently, the number of tasks in F is even. □

6.6 Scheduling in the Order of Modified Levels is Optimal

Once the modified levels of tasks are determined, it is easy to compute an optimal schedule, as we will show next. First, we sort all tasks that have not been moved in nonincreasing order of their (modified) level. On each level, tasks are ordered by their preorder number in the decomposition tree. Then, the moved tasks are inserted into the resulting sequence such that each moved task is the last task on its modified level. This is possible, since each level receives at most one moved task, by Lemma 6.10. Let σ denote this sequence, and let $pos(x)$ be the position of task x in σ.

Theorem 6.14 *The mapping S that maps each task x to timestep $\lceil pos(x)/2 \rceil$ is an optimal two processor schedule for $(T, \prec)$.*

Proof. Assume for the time being that S is a valid schedule. Let B be the original decomposition tree. We have added $2\omega(root(B)) - |root(B)|$ empty tasks to the original task system. Hence,

$$length(S) = \left\lceil \frac{|T| + 2\omega(root(B)) - |root(B)|}{2} \right\rceil = \omega(root(B)).$$

Since $\omega(root(B))$ is a lower bound for the length of an optimal two processor schedule, we obtain that no schedule for $(T, \prec)$ can be shorter than S. Hence, S

is optimal. (As a byproduct, this proves our claim that $\omega(root(B))$ is the length of an optimal schedule.)

What remains to show is that S is a valid schedule, *i.e.*, S obeys the precedence constraints. Let x be a predecessor of y. By Lemma 6.12, $mlevel(x) > mlevel(y)$. Since tasks are scheduled in nonincreasing order of modified levels, it holds that $S(x) \leq S(y)$. Clearly, no precedence constraints are violated if $S(x) < S(y)$. Therefore, assume that $S(x) = S(y)$.

On each (modified) level there is exactly one backbone task. Since σ consists of tasks ordered by nonincreasing modified level and x and y are adjacent in σ, their modified level differs by exactly one.

Assume y has been moved. Then y is the only task on its modified level, because in σ, for each modified level, the moved task is behind the other tasks. A contradiction, since at least one other task, namely a backbone task, is on the same modified level as y.

Assume x has been moved. Since y has not been moved, its modified level is its original level. But the original level of x is lower than its modified level. Furthermore, $mlevel(x) = mlevel(y) + 1$. Consequently, $level(x) < level(y) + 1$ and hence, x can not be a predecessor of y. A contradiction.

Since y is the first task in σ on its modified level, all other tasks on this level that are not moved have a larger preorder number in the decomposition tree than y. By Lemma 6.1, y must be a backbone task.

Since x is not a moved task but x is the last in σ on its modified level, we obtain that there are no moved tasks on the modified level of x. Hence, all tasks on the modified level of x have a smaller preorder number than x in the decomposition tree. Since y is successor of x, we obtain that y is successor of all tasks on the modified level of x, by Lemma 6.2. As a consequence, all tasks to the left of y in σ that are not moved tasks are predecessors of y, because every such task has a successor on the modified level of x. Hence, all tasks on a modified level higher than that of y are either predecessors of y or moved tasks. We apply Lemma 6.13 and obtain that the number of tasks to the left of y in σ is even. Since x is to the left of y in σ, x and y can not be mapped to the same timestep. A contradiction. Hence, no precedence constraints are violated in S. We conclude that S is an optimal two processor schedule for $(T, \prec)$. □

6.7 An Efficient Parallel Implementation

In this section we describe how the algorithm given in the last sections can be implemented on a parallel machine. Let us first summarize the algorithm. We assume that a decomposition tree $B = (V,E)$ for G is given.

1. We compute the height of $G(v)$, for all $v \in V$. For each "P" vertex v in B, we exchange the left and right child of v if $\mathit{height}(G(l(v))) < \mathit{height}(G(r(v)))$.

2. We determine all vertices with an ancestor that is the right child of a "P" vertex in B. Those vertices belong to the set $\beta(B)$.

3. We compute $|v|$ and $\omega(v)$, for all vertices in B.

4. We construct the extended decomposition tree $\hat{B} = (\hat{V},\hat{E})$ by adding $2\omega(\mathit{root}(B)) - |\mathit{root}(B)|$ empty tasks to B, using parallel compositions. Furthermore, we determine $|v|$ and $\omega(v)$, for all new vertices v in $\hat{B}$, and we determine $\beta(\hat{B})$.

5. We give each vertex in $\hat{B}$ a parenthesis label and compute the parenthesis word of the root of $\hat{B}$. Then we determine the matching pairs in this word.

6. We compute the levels of tasks and assign a modified level to each vertex in $\beta(\hat{B})$. Then we compute the modified levels of the tasks not in $\beta(\hat{B})$ using the matching pairs in $\pi(\mathit{root}(\hat{B}))$.

7. We sort all unmoved tasks by nonincreasing level such that tasks on the same level are sorted by their preorder number in $\hat{B}$. Then each moved task x is inserted into the resulting sequence such that x is the last task on its (modified) level. Let σ denote the sequence that we obtain.

8. Finally, we output the schedule that maps the i-th task of σ to timestep $\lceil i/2 \rceil$.

In the following we show that each step of the algorithm can be performed efficiently in parallel.

Step (1). According to Theorem 3.22, the height of all defining subgraphs of G with regard to B can be determined on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors. Checking whether $G(r(v))$ is higher than $G(l(v))$ and exchanging the left and right child if necessary can be done for all vertices in constant time with n processors or in $O(\log n)$ time with $n/\log n$ processors.

Step (2). To determine the vertices of $\beta(B)$, we use the Euler-Tour technique. We split the decomposition tree into a number of trees and apply the Euler-tour technique to each of these trees separately (but simultaneously).

The trees are found as follows. Each vertex v in B that is the right child of a “P” vertex becomes the root of a tree, namely, its subtree. If there is a vertex w in the subtree rooted at v that itself becomes a root of a tree (because it is the right child of a “P” vertex), then w and its subtree is excluded from v’s tree. We end up with a number of disjoint trees each having a root that is the right child of a “P” vertex in the decomposition tree. Then, we mark the vertices contained in these trees, using the Euler-tour technique. In the end, the set $\beta(B)$ consists of the unmarked vertices.

Steps (3)+(4). We compute $|v|$ and then $\omega(v)$ using tree contraction on the decomposition tree. We require similar vertex functions as used in the computation of the heights of the defining subgraphs of G with regard to B (cf. Subsection 3.11.2). Given $|root(B)|$ and $\omega(root(B))$, it is not difficult to construct an extended decomposition tree $\hat{B}$ from B, and to determine $|v|$ and $\omega(v)$ for all new vertices v in $\hat{B}$.

Step (5). Consider a path p that starts at the root vertex of $\hat{B}$ and visits each vertex in the following way. Let v denote the current vertex. If v is an “S” vertex in $\beta(\hat{B})$, then the path visits the vertices in the right subtree, returns from there, then visits the vertices in the left subtree, and then returns to the parent of v. If v is a leaf, then the path immediately returns to the parent of v. In all other cases, the path visits the vertices in the left subtree of v first, returns from there, then visits the vertices of the right subtree, and finally returns to the parent of v. Note that this path is very similar to the Euler-tour of the extended decomposition tree, except at “S” vertices in $\beta(\hat{B})$, where the path visits the right subtree first. Since $\hat{B}$ is binary, the path p can be constructed on n processors in constant time. Using p, we link the parenthesis labels of the vertices in $\hat{B}$ into a list that contains the labels in the same order as the parenthesis word of the root of $\hat{B}$. From this list, the parenthesis word of $root(\hat{B})$ can easily be computed using list ranking and prefix operations. To compute the matching pairs in $\pi(root(\hat{B}))$, we use the parenthesis matching algorithm given in Section 3.9.

Step (6). To compute the level of each task in the precedence graph, we use the Euler-tour technique on $\hat{B}$. In Section 3.7, the Euler-tour of a tree followed the contour of the tree counterclockwise. Here, we let the Euler-tour run clockwise along the contour of the decomposition tree. In particular, the Euler-tour starts with a downgoing arc from the root of $\hat{B}$ to its right child. For every leaf w, we associate with the downgoing arc $d(v,w)$ a value of 1. For every “P” vertex v with right child w, we associate with the upgoing arc $u(v,w)$ a value of $-height(G(w))$. All other arcs in the Euler-tour obtain value 0. Then for each leaf w, the prefix sum of the arc values on the Euler-tour up to the downgoing arc $d(v,w)$ is the level of task w in the precedence graph. (Recall that $G(l(v))$ is at least as high as $G(r(v))$.) Given the levels of all tasks and the matching pairs in $\pi(root(\hat{B}))$, it is not difficult to compute the modified levels. In particular, the modified levels of "P" vertices in $\beta(\hat{B})$ are already computed. Consider the prefix sums of arc values obtained in the computation of the task levels, and let $d(v,w)$ be the downgoing arc from a "P" vertex v in $\beta(\hat{B})$ to its right child w. If the original value associated with $d(v,w)$ is zero, then the prefix sum obtained for $d(v,w)$ plus 1 corresponds to the lowest level of a task in the subtree rooted at v. If the original value associated with $d(v,w)$ is 1 (in this case w is a leaf), then the prefix sum obtained for $d(v,w)$ is the desired value.

Steps (7)+(8). Let $\hat{G}$ denote the task system that $\hat{B}$ represents. Let $\hat{G}^R$ denote the graph that we obtain from $\hat{G}$ by reversing the direction of all edges. Let v be a vertex of $\hat{G}$ (and $\hat{G}^R$). We observe that the level of v in $\hat{G}$ is the depth of v in $\hat{G}^R$. Hence, if we sort tasks by nondecreasing depth in $\hat{G}^R$, we sort tasks by nondecreasing level in $\hat{G}$ at the same time. To obtain a decomposition tree for $\hat{G}^R$, we exchange the left and right child of each vertex in the decomposition tree for $\hat{G}$. Let $\hat{B}^R$ denote the resulting tree. In Section 3.11, we describe how the breadth-first traversal of a VSP dag with regard to a given decomposition tree can be computed on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors. We apply this algorithm to $\hat{B}^R$ and obtain a list of all tasks ordered by nondecreasing level (in $\hat{G}$) such that tasks on the same level are ordered by their preorder number in $\hat{B}^R$. We reverse this list and obtain a list σ of all tasks ordered by nonincreasing level such that tasks on the same level are ordered by their preorder number in $\hat{B}$. (Note that the breadth-first traversal of $\hat{B}^R$ restricted to leaves is the reversed breadth-first traversal of $\hat{B}$ restricted to leaves.) Then, each moved task is dropped from σ and reinserted such that it becomes the last task in σ on its modified level. Clearly, this can be done using prefix operations. Finally, tasks are mapped to timesteps according to their position in σ. By Theorem 6.14, this schedule is optimal.

We conclude that all operations necessary to compute an optimal two processor schedule can be performed on $n/\log n$ EREW PRAM processors in time $O(\log n)$, provided that a decomposition tree is given. We can summarize:

Theorem 6.15 *Let $(T,\prec)$ be a series parallel order consisting of n tasks, and let G be a series parallel precedence graph for $(T,\prec)$. Given a decomposition tree for G, an optimal two processor schedule for $(T,\prec)$ can be computed on the EREW PRAM in time $O(\log n)$ using $n/\log n$ processors.*

By Theorem 3.21, the decomposition tree of a VSP dag can be computed on the EREW PRAM in time $O(\log(n+e)+\log n \log^* n)$ and on the CRCW PRAM in time $O(\log(n+e))$, where in both cases the number of operations is $O(n+e)$. We obtain:

Corollary 6.16 *Let $(T,\prec)$ be a series parallel order, given as a series parallel precedence graph with e edges and n vertices. Then an optimal two processor schedule for $(T,\prec)$ can be computed on the EREW PRAM in time $O(\log(n+e)+\log n \log^* n)$ and on the CRCW PRAM in time $O(\log(n+e))$. The work performed is $O(n+e)$.*

CHAPTER 7

Scheduling Interval Orders with Communication Delays

In the last three chapters, we have proposed efficient parallel algorithms that optimally schedule unit length tasks. These algorithms work well if either no data needs to be exchanged between tasks or the time necessary to communicate the result of a task to its successor tasks is negligible. In the past few years, scheduling with communication delays has received considerable attention. In this extended setting, the schedule additionally takes into account the time required to communicate data between processors. More precisely, after finishing a task, some time must pass before any of its successors can be started on a different processor, in order to allow for transportation of results from one task to the other. If the successor task is executed on the same processor, then no delay is needed unless it is necessary to wait for results from some other processor.

We are interested in computing schedules of minimal length for unit execution time tasks subject to unit communication delays, *i.e.*, all tasks have equal length and the same amount of time is required for interprocessor communication. Furthermore, we assume that no task duplication is allowed. If precedence constraints can be arbitrary and the number of processors is part of the problem instance, then this problem is $\mathcal{NP}$-complete [RS87]. Surprisingly, the problem remains $\mathcal{NP}$-hard even if precedence constraints are restricted to be trees [LVV93]. For some other special cases, polynomial solutions have been found. This includes scheduling interval ordered tasks [Pic92, AER95], scheduling trees on two processors [Pic92, LVV93, GT93, VRKL96], scheduling trees on any constant number of processors [VRKL96], and scheduling series parallel graphs on two processors [FLMB96].

In this chapter, we focus on precedence constraints given by interval orders. Interval ordered tasks occur in a variety of manufacturing problems and the literature on interval orders is quite rich [FG65, Gav72, Fis85, Kle93]. A sequential algorithm for scheduling interval orders was given in [PY79]. Algorithms for scheduling interval orders with communication delays can be found in [Pic92] and [AER95]. All these algorithms are based on list scheduling and appear to be inherently sequential. In fact, list scheduling has been shown to be $\mathcal{P}$-complete in similar contexts [DUW86, HM87b] and one might be tempted to conjecture that $\mathcal{NC}$ algorithms for scheduling interval orders based on list scheduling do not exist as well. On the other hand, fast parallel algorithms have been found that compute optimal schedules for interval orders, albeit by rather different methods [SH93, May96]. Actually, the $\mathcal{NC}$ algorithms in [SH93, May96] compute a schedule identical to the list schedule in [PY79].

In the sequel, we first show that there exists a linear time sequential algorithm that optimally schedules interval orders with communication delays, improving on results given in [AER95]. Our second and main contribution in this chapter is an $\mathcal{NC}$ algorithm for the problem. If implemented on an EREW PRAM this algorithm runs in time $O(\log^2 n)$ using $n^3/\log n$ processors. If implemented on a CRCW PRAM it requires only $O(\log n)$ time and uses n^3 processors. Our parallel algorithm proceeds in two stages. In stage one, we determine for every task the number of timesteps required to schedule all of its predecessors in the interval order. These "scheduling distances" provide us with information on the structure of the desired optimal schedule. In stage two, we use this information to construct an instance of a different scheduling problem where tasks are not constrained by a precedence relation but have individual release times and deadlines. We compute an optimal schedule for this instance using a parallel algorithm known from the literature, and thus obtain an optimal schedule for the original problem.

The remainder of this chapter is organized as follows. In the next section, we introduce concepts related to scheduling with unit execution time tasks and unit communication times. In the section thereafter, we review interval orders and define basic terms used in our algorithm. In Section 7.3, we present a linear time sequential algorithm for scheduling interval orders with communication delays. In the section thereafter we outline the parallel algorithm. Then, we derive a lower bound on the length of optimal schedules, and in Section 7.6, we analyze the structure of optimal schedules. These results are used in Section 7.7 to show that the scheduling distance of each task, *i.e.*, the length of an optimal schedule for all of its predecessors, corresponds to a longest path in a "distance graph", and that it can be computed efficiently in parallel. We then describe the remaining part of the algorithm where we construct the task system with release times and deadlines and output an optimal schedule for this task system.

7.1 UECT Scheduling

An instance of the unit execution time and unit communication time (UECT) scheduling problem consists of a task system $(T,\prec)$ and a number m of identical target processors, where T is a set of n tasks and $\prec$ is a partial order on this set of tasks. An *m-processor UECT schedule* for $(T,\prec)$ is a mapping S of T to positive integer timesteps such that

1. $x \prec y$ implies $S(x) < S(y)$,
2. no more than m tasks are mapped to the same timestep,
3. at most one successor of every task x is mapped to $S(x)+1$, and
4. at most one predecessor of every task x is mapped to $S(x)-1$.

Given a UECT schedule S, it is not difficult (neither sequentially nor in parallel) to determine an assignment of tasks to processors such that two tasks x and y are assigned to the same processor if $S(y) = S(x)+1$ and $x \prec y$. At the end of this chapter, we will show how such an assignment can be computed in parallel for the case of interval orders.

With regard to a given UECT schedule, we say that a task x is *ready* at timestep τ if all predecessors of x are mapped to timesteps earlier than τ and at most one predecessor of x is mapped to $\tau-1$. Task x is *available* at timestep τ if x is ready and there is no task y mapped to timestep τ such that x and y have a common predecessor in timestep $\tau-1$.

7.2 Interval Orders and Interval Representations

We assume that the partial order $\prec$ is given as a precedence graph. Recall that $\prec$ is an *interval order* on T iff, for all $t,t',x,y \in T$, $t \prec t'$ and $x \prec y$ implies that $t \prec y$ or $x \prec t'$. We obtain a more intuitive characterization of interval orders by using intervals on the real line. A partial order $\prec$ on T is an interval order iff there exists a mapping I from T to closed intervals on the real line such that $I(x)$ is completely to the left of $I(y)$ iff $x \prec y$. The mapping I is called an *interval representation* of the interval order $\prec$. Figure 7.1 shows a precedence graph of an interval order containing fourteen tasks. To the right of the precedence graph, an interval representation of the interval order is depicted.

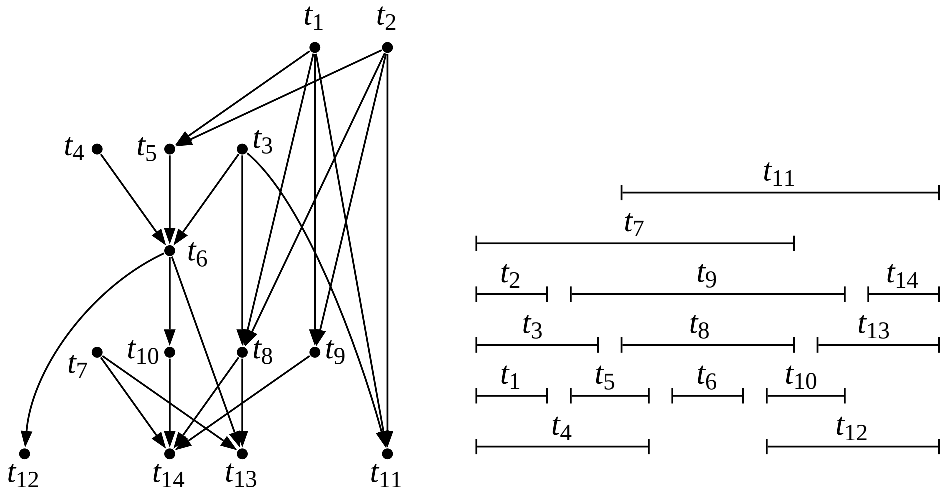
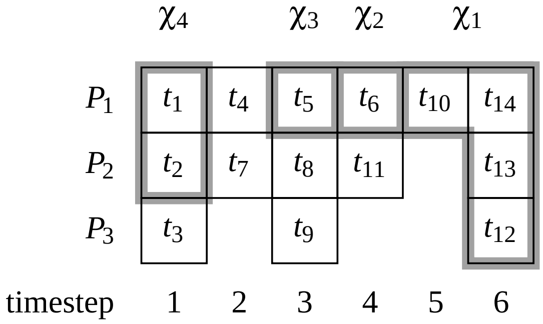


Figure 7.1: *A precedence graph of an interval order and an interval representation of it.*

Figure 7.2 shows a 3-processor UECT schedule for the interval order given in Figure 7.1. In this schedule, timesteps 2, 4, and 5 are partial. Task t_9 is not ready at timestep 2 because two predecessors of t_9 are scheduled in timestep 1. On the other hand, t_{12} is ready at timestep 5 but it is not available. This is because t_{10} is mapped to timestep 5, and t_6, which is a common predecessor of t_{10} and t_{12}, is mapped to timestep 4.

Let $(T, \prec)$ be some interval order, and let $x \in T$. Then $N(x)$ denotes the set of proper successors of x, *i.e.*, $N(x) = \{y | x \prec y\}$. Similarly, $P(x)$ is the set $\{y | y \prec x\}$ of (proper) predecessors of x. Note that $x \notin N(x)$ and $x \notin P(x)$. Since successor sets of tasks can be defined in terms of right endpoints of intervals in interval representations of $\prec$, we can easily see that for every pair x, y of tasks the successor sets of x and y are comparable with respect to set inclusion, *i.e.*, either $N(x) \subset N(y)$, or $N(x) = N(y)$, or $N(y) \subset N(x)$. Hence,



P_1	t_1	t_4	t_5	t_6	t_{10}	t_{14}
P_2	t_2	t_7	t_8	t_{11}		t_{13}
P_3	t_3		t_9			t_{12}
timestep	1	2	3	4	5	6

Figure 7.2: *A 3-processor UECT schedule for the given interval order.*

there exist total orders on T that are compatible with the order on successor sets. For instance, the tasks in Figure 7.1 could be ordered as t_1, t_2, t_3, t_5, t_4, t_6, t_8, t_7, t_9, t_{10},t_{14},t_{13},t_{12},t_{11} since

$$N(t_1) = N(t_2) \supset N(t_3) \supset N(t_5) = N(t_4) \supset N(t_6) \supset N(t_8) = \\ N(t_7) \supset N(t_9) = N(t_{10}) \supset N(t_{14}) = N(t_{13}) = N(t_{12}) = N(t_{11}).$$

Note that for every task x and every list L of all tasks ordered by nonincreasing (size of) successor set, the set $P(x)$ is a prefix of L. For example, in the above list, $P(t_{10})$ is the prefix up to t_6.

By $Q(x)$ we denote the set $\{y|N(x) \subseteq N(y)\}$ consisting of all tasks whose successor set contains $N(x)$. For example, $Q(t_5) = \{t_1,t_2,t_3,t_4,t_5\}$. Note that if x is a predecessor of y with minimal successor set, then $Q(x) = P(y)$. Furthermore, every task in $Q(x)$ is a predecessor of all tasks in $N(x)$.

7.3 A Sequential Algorithm

In the context of unit execution times and unit communication delays list scheduling works as follows. We are given a list of all tasks that determines for each task its priority. Assume that the first $\tau - 1$ timesteps have already been processed, that is, tasks have been mapped to them and these tasks have been removed from the list. To find the tasks for timestep τ, perform the following. Scan the list from left to right and pick a task available at timestep τ, remove it from the list, and map it to timestep τ. Repeat this until no more unscheduled tasks are available at timestep τ or m tasks have been mapped to τ. If there are still tasks in the list, repeat the above process for timestep $\tau + 1$. Otherwise, we are finished, having obtained a UECT schedule of length τ.

Theorem 7.1 ([Pic92]) *Let* $(T, \prec)$ *be an interval order, and let L be a list of all tasks, sorted by nonincreasing successor set. Then the UECT list schedule for L is an optimal UECT schedule for* $(T, \prec)$.

For example, the schedule depicted in Figure 7.2 is the UECT list schedule constructed from the list t_1, t_2, t_3, t_5, t_4, t_6, t_8, t_7, t_9, t_{10}, t_{14}, t_{13}, t_{12}, t_{11}. By Theorem 7.1, this schedule is optimal since the tasks in the list are ordered by nonincreasing successor set.

In the following we briefly discuss how to construct a UECT list schedule for a list L in time linear in the size of the given precedence graph (for a similar problem, see [Set76]). We assume that in L all predecessors of a task x appear before x. We scan the list once from left to right and map each task to the earliest possible timestep. We claim that the resulting schedule is identical to the list schedule obtained by the naïve implementation of the algorithm given at the beginning of this section. To see this, consider some prefix L' of L. Let S be the UECT list schedule for L, and let S' be the UECT list schedule for L'. We observe that S restricted to the tasks in L' is identical to S'. This is because no task in $L - L'$ that is scheduled by S in a timestep $\leq$ *length*(S') does influence how the list scheduling algorithm schedules the prefix L'. Hence, in order to determine the timestep to which x gets mapped in the list schedule, it suffices to compute the list schedule S' for the prefix of L that ends right before x and to determine the earliest partial timestep at which x is available in S'.

To find the earliest timestep at which x is available, we have to check the immediate predecessors of x (recall that all predecessors of x appear before x in L). More precisely, we have to determine the latest timestep τ with a predecessor of x, and we have to determine the number of predecessors of x mapped to τ. If more than one predecessor of x is mapped to τ, then x is available at timestep $\tau + 2$. If there is only one predecessor y of x mapped to τ, then we have to check whether a successor of y is already mapped to timestep $\tau + 1$. If this is the case, then x is available at timestep $\tau + 2$. Otherwise, x is available at timestep $\tau + 1$. In order to be able to check whether y has a successor in timestep $\tau + 1$, some bookkeeping is required. For each task y, we maintain a boolean variable that is initially *false* and is set to *true* when an immediate successor of y gets mapped to timestep $S(y) + 1$.

To determine the earliest *partial* timestep at which x is available, we use a *union-find* data structure. A UECT list schedule has length at most n. Hence, the only relevant timesteps are those between $1, \ldots, n$. Initially, all of them are marked “partial”, and we let each of them form a distinct set in the union-find structure. When some timestep τ becomes full during scheduling, we join the set in the union-find structure to which τ belongs with the set to which timestep $\tau + 1$ belongs. We call the latest timestep in a set the *canonical element*. As a consequence, the canonical element of each set is always a partial timestep. To find the earliest partial timestep at which x is available, we simply have to find the canonical element of the set that contains the earliest timestep at which x is available.

Theorem 7.2 *Let $(T, \prec)$ be an interval order given as a (not necessarily transitively closed) precedence graph G, consisting of n tasks and e edges. An optimal m-processor UECT schedule for $(T, \prec)$ can be computed in $O(n+e)$ time.*

Proof. In [Gab81], an algorithm is presented that checks in time $O(n+e)$ whether a given (not necessarily transitively closed) precedence graph is the precedence graph of an interval order. If this is the case, then the algorithm determines the sizes of all successor sets. Given these numbers, we can sort tasks by nonincreasing successor set in time $O(n)$, using bucket sort. Finally, we apply our second list scheduling algorithm to the resulting list, and obtain an optimal UECT schedule for $(T, \prec)$.

For each task x, our algorithm requires time linear in the indegree of x in G to determine the earliest timestep at which x is available. As a consequence, we need $O(n+e)$ time to determine all earliest timesteps. In order to find the earliest *partial* timesteps, at most one union operation and at most one find operation is performed for each task. We use the algorithm for static union-find given in [GT85] and require $O(n)$ time to perform all unions and finds. □

7.4 Outline of the Parallel Algorithm

Let $(T, \prec)$ be an interval order, and let L be a list of all tasks, ordered by nonincreasing successor set. Our parallel algorithm computes a schedule that is identical to the UECT list schedule S for L. If $m = 1$, then S maps the i-th task of L to timestep i. Clearly, this schedule can easily be computed in parallel. In the rest of this chapter, we assume $m > 1$. The algorithm proceeds as follows:

1. For each task x, we determine the latest timestep in S with a predecessor of x. Let $\tau(x)$ denote this timestep. Furthermore, we determine whether one predecessor or more than one predecessor of x is mapped to $\tau(x)$. We

call x *favored task candidate* of timestep $\tau(x)$, if exactly one predecessor of x is mapped to $\tau(x)$.

2. For each timestep τ, we determine the leftmost favored task candidate of τ in L. This task is called *favored task* of timestep τ in L.

3. We give each task x a release time $r(x)$ as follows: we set $r(x)$ to 1 if x has no predecessors, we set $r(x)$ to $\tau(x)+1$ if x is favored task of timestep $\tau(x)$ in L, and otherwise, we set $r(x)$ to $\tau(x)+2$.

4. We give each task x a deadline $d(x)$, where $d(x)$ is set to $|T|$ plus the position of x in L.

5. We drop all precedence constraints and compute an earliest deadline schedule for the task system (T,r,d). Finally, we output the resulting schedule.

In the next three sections we are concerned about step 1. We start in the following section by giving a lower bound on the length of a UECT schedule.

7.5 UECT Packings

Let $U_r,\dots,U_1$ be pairwise disjoint nonempty subsets of tasks such that every task in U_i is predecessor of all tasks in U_{i-1}, for $i=2,\dots,r$. We want to determine the length of a shortest possible schedule for $U_r,\dots,U_1$, ignoring all precedence constraints between tasks inside a set U_i. We schedule one set after the other in a greedy fashion, starting with U_r. Let τ_i denote the latest timestep that contains a task of U_{i+1}, and let k_i denote the number of tasks of U_{i+1} mapped to τ_i. The tasks of U_r are mapped to the first $\lceil |U_r|/m \rceil$ timesteps such that none of these timesteps, except possibly the last, is partial. It follows that $|U_r| \bmod m$ tasks are mapped to timestep τ_{r-1}, if $|U_r| \bmod m > 0$. If $|U_r| \bmod m$ is zero, then m tasks are mapped to τ_{r-1}.

Definition 7.3 *Let s be a positive integer. Then*

$$s \operatorname{mod}_1 m := ((s-1) \bmod m) + 1.$$

Hence, $|U_r| \operatorname{mod}_1 m$ is the number of tasks of U_r mapped to the latest timestep that contains a task of U_r, *i.e.*, $k_{r-1} = |U_r| \operatorname{mod}_1 m$. Now assume that we have already scheduled $U_r,\dots,U_{i+1}$ in a greedy fashion. Then the next set U_i is scheduled as follows. If $k_i = 1$, then we map exactly one task of U_i to $\tau_i + 1$, and we map all other tasks of U_i to timesteps $\tau_i+2,\ldots,\tau_i+1+\lceil(|U_i|-1)/m\rceil$ such that none of these timesteps, except possibly the last, is partial. It follows that the latest timestep with a task of U_i is $\tau_{i-1}=\tau_i+1+\lceil(|U_i|-1)/m\rceil$. If $|U_i|>1$, then $(|U_i|-1)\operatorname{mod}_1 m$ tasks are mapped to τ_{i-1}, *i.e.*, $k_{i-1}=(|U_i|-1)\operatorname{mod}_1 m$. If $|U_i|=1$, then one task is mapped to τ_{i-1}, *i.e.*, $k_{i-1}=1$. On the other hand, if $k_i>1$, then no task of U_i can be mapped to τ_i+1, due to communication delays. In this case, we map all tasks of U_i to timesteps $\tau_i+2,\ldots,\tau_i+1+\lceil|U_i|/m\rceil$ such that none of these timesteps, except possibly the last, is partial. Hence, $k_{i-1}=|U_i|\operatorname{mod}_1 m$ tasks are mapped to timestep $\tau_{i-1}=\tau_i+1+\lceil|U_i|/m\rceil$. We repeat the above process until all sets are scheduled. We call this kind of schedule an *m-processor UECT packing* of $U_r,\ldots,U_1$.

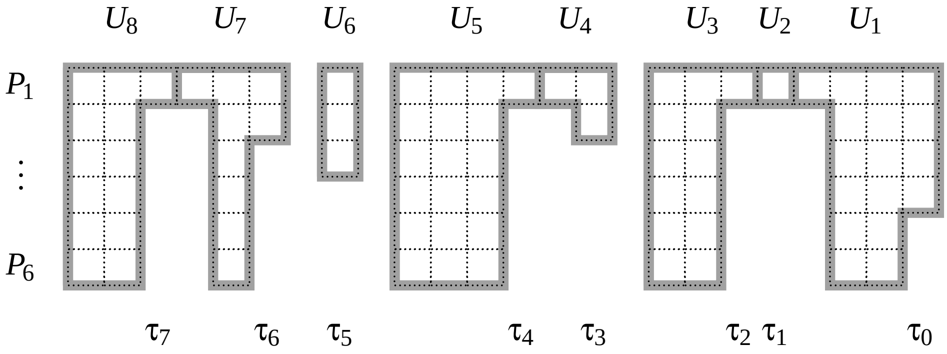


Figure 7.3: *A 6-processor UECT packing.*

For example, consider eight sets $U_8,\ldots,U_1$ of sizes 13, 9, 3, 19, 3, 13, 1, and 17. The 6-processor UECT packing of these sets is shown in Figure 7.3, where tasks are depicted as dotted squares and each set is framed in grey. The length of this UECT packing is 24.

Definition 7.4 *Let $U_r,\ldots,U_1$ be nonempty subsets of T such that every task in U_i precedes all tasks in U_{i-1}, for $i=2,\ldots,r$. Let k denote the number of tasks mapped to the last timestep in an m-processor UECT packing of $U_r,\ldots,U_1$. The* packing interface *of $U_r,\ldots,U_1$, denoted by $\phi(U_r,\ldots,U_1)$, is defined as follows:*

$$\phi(U_r,\ldots,U_1):=\begin{cases}1 & \textit{if } k=1,\\ 0 & \textit{otherwise.}\end{cases}$$

Given the packing interface of $U_r,\ldots,U_{i+1}$, the packing interface of $U_r,\ldots,U_i$ can be computed using the following function:

Definition 7.5 *Let $a \in \{0,1\}$, and let U be a nonempty subset of T. We define*

$$append(a,U) := \begin{cases} 1 & \textit{if } |U| = 1 \textit{ or } (|U| - a) \bmod_1 m = 1, \\ 0 & \textit{otherwise.} \end{cases}$$

Then, $\phi(U_r,\ldots,U_i) = append(\phi(U_r,\ldots,U_{i+1}),U_i)$, for $i = 1,\ldots,r-1$, and

$$\phi(U_r) = \begin{cases} 1 & \text{if } |U_r| \bmod_1 m = 1, \\ 0 & \text{otherwise.} \end{cases}$$

For $i > 0$, $\phi(U_r,\ldots,U_{i+1})$ is the number of tasks of U_i mapped to $\tau_i + 1$. As a consequence, τ_{i-1} equals $\tau_i + 1 + \lceil (|U_i| - \phi(U_r,\ldots,U_{i+1}))\,/\,m \rceil$. Let $length(U_r,\ldots,U_1)$ denote the length τ_0 of an m-processor UECT packing of the sets $U_r,\ldots,U_1$. This length can now be written as

$$length(U_r,\ldots,U_1) = \left\lceil \frac{|U_r|}{m} \right\rceil + \sum_{i=1}^{r-1} \left(1 + \left\lceil \frac{|U_i| - \phi(U_r,\ldots,U_{i+1})}{m} \right\rceil \right) \qquad (7.1)$$

and we can easily derive the following lower bound on the length of an optimal UECT schedule:

Lemma 7.6 *Let $(T,\prec)$ be some task system, and let opt be the minimal length of an m-processor UECT schedule for $(T,\prec)$. Let $U_r,\ldots,U_1$ be pairwise disjoint nonempty subsets of T such that every task in U_i precedes all tasks in U_{i-1}, for $i = 2,\ldots,r$. Then opt $\geq length(U_r,\ldots,U_1)$.*

Proof. Any UECT schedule for $(T,\prec)$ is at least as long as a UECT packing of $U_r,\ldots,U_1$. The length of such a packing is given by (7.1). □

7.6 Block Decompositions

The lower bound given in the last section holds for task systems with arbitrary precedence constraints. In this section we prove that in the case of interval orders we can always find disjoint subsets of tasks such that the length of a UECT packing of these sets equals the length of an optimal UECT schedule for the interval order.

Definition 7.7 *Let $(T,\prec)$ be an interval order, and let opt be the minimal length of an m-processor UECT schedule for $(T,\prec)$. Let $\chi_r,\ldots,\chi_1$ be pairwise disjoint nonempty subsets of T such that*

1. *there exist tasks* $t_0, \ldots, t_{r-1}$, *with* $\chi_r = Q(t_{r-1})$, $\chi_i = N(t_i) \cap Q(t_{i-1})$ *for* $i = 1, \ldots, r-1$, *and* $Q(t_0) = T$,

2. $opt = length(\chi_r, \ldots, \chi_1)$.

Then $\chi_r, \ldots, \chi_1$ *is an* m-processor UECT block decomposition *for* $(T, \prec)$ *and each set* χ_i *is called a* block.

Note that in a UECT block decomposition $\chi_r, \ldots, \chi_1$, every task in χ_i precedes all tasks in χ_{i-1}, for $i = 2, \ldots, r$. Block decompositions with similar properties have been studied by Coffman and Graham [CG72], by Helmbold and Mayr [HM87b, May96], and in Chapter 5.

A 3-processor UECT block decomposition for the interval order in Figure 7.1, consisting of four blocks $\chi_4, \ldots, \chi_1$, is depicted in Figure 7.2. In the proof of the following theorem, we show how to construct such a decomposition.

Theorem 7.8 *For every interval order, there exists an m-processor UECT block decomposition.*

Proof. Let $(T, \prec)$ be an interval order, let L be a list of all tasks sorted by nonincreasing successor set, and let S be the UECT list schedule for L. The first timestep occupied by a task in S is timestep 1. For ease of presentation, we introduce a new task t_{top} that precedes all tasks in T and is mapped to timestep -1. Let τ_0 be the last timestep of S, and let T_0 denote the set of tasks scheduled in timestep τ_0. Furthermore, let t_0 be an arbitrary task in T_0. We define inductively, as long as $\tau_{i-1} > 0$:

$\tau_i :=$ either $\tau_{i-1} - 1$, if some task is scheduled in $\tau_{i-1} - 1$ that precedes all tasks in T_{i-1}, or otherwise the latest timestep τ before $\tau_{i-1} - 1$ such that $\tau + 1$ is partial or a task u is scheduled in timestep $\tau + 1$ with $N(u) \subset N(t_{i-1})$.

$\tilde{t}_i :=$ some task scheduled in timestep τ_i with maximal successor set.

$\chi_i :=$ $N(\tilde{t}_i) \cap Q(t_{i-1})$.

$T_i :=$ the set of tasks scheduled in timestep τ_i that precede all tasks in χ_i.

$t_i :=$ some task in T_i with minimal successor set.

Let $\chi_r, \ldots, \chi_1$ be the sets defined by this algorithm. Then $\tau_r = -1$ and $\tilde{t}_r = t_r = t_{top}$. We claim that $\chi_r, \ldots, \chi_1$ is a UECT block decomposition for $(T, \prec)$. We first prove that $\chi_r, \ldots, \chi_1$ are well defined. To this end, we have to show that $\tilde{t}_i$ and t_{i-1} are well defined, for $i = 1, \ldots, r$. If $\tilde{t}_i$ exists, then t_i exists

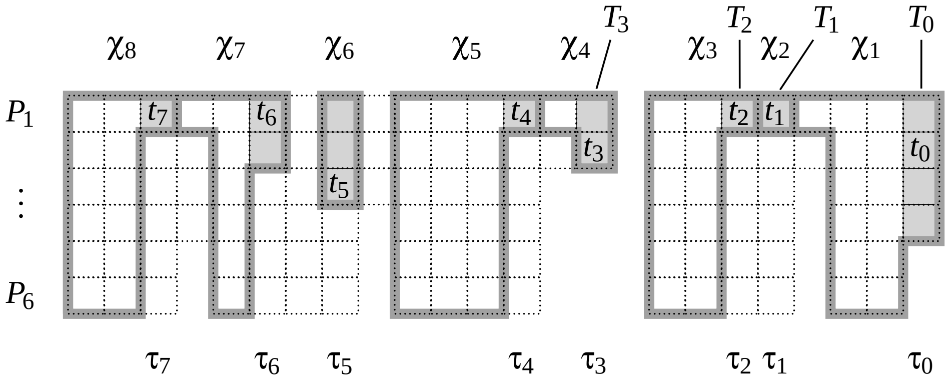


Figure 7.4: *We construct blocks from right to left by inductively defining appropriate timesteps τ_i, sets T_i, and tasks t_i (see proof of Theorem 7.8). Tasks are depicted as dotted squares, each set T_i is filled grey, and each block is framed in grey.*

as well, because T_i contains $\tilde{t}_i$. Assume i is the first index for which $\tilde{t}_i$ can not be defined. This implies that no task is mapped to timestep τ_i. In this case, $\tau_i \leq \tau_{i-1} - 2$, by construction. We have to consider two cases.

case 1: $\tau_i = \tau_{i-1} - 2$. Timestep $\tau_i + 1$ is partial or a task u is mapped to $\tau_i + 1$ with $N(u) \subset N(t_{i-1})$. The set T_{i-1} is not empty, since it contains t_{i-1}. By construction, $N(t_{i-1}) \subseteq N(x)$, for all $x \in T_{i-1}$, since t_{i-1} has minimal successor set among tasks in T_{i-1}. Hence, if u exists, then $N(u) \subset N(x)$, for all $x \in T_{i-1}$, *i.e.*, all tasks in T_{i-1} have higher priority in L than u. Since no task is mapped to τ_i, the only possible reason why the list scheduling algorithm did not map a task of T_{i-1} to $\tau_i + 1$ is that every $x \in T_{i-1}$ has a predecessor in timestep $\tau_i + 1$. By definition of interval orders, there must exist a task in $\tau_i + 1$ $(= \tau_{i-1} - 1)$ that precedes all tasks in T_{i-1}. We obtain a contradiction, since τ_i should equal $\tau_{i-1} - 1$, by construction.

case 2: $\tau_i < \tau_{i-1} - 2$. Let A denote the set of tasks mapped to $\tau_i + 2$. By construction, $|A| = m$ and for all $x \in A$, $N(t_{i-1}) \subseteq N(x)$. On the other hand, $\tau_i + 1$ is partial or a task u is mapped to $\tau_i + 1$ with $N(u) \subset N(t_{i-1})$. Therefore, all tasks in A have higher priority in L than u, if u exists. The reason why the list scheduler did not map a task of A to $\tau_i + 1$ is that every $x \in A$ has a predecessor in timestep $\tau_i + 1$. By definition of interval orders, there must exist a task in $\tau_i + 1$ that precedes all tasks in A. But this violates communication delays, since there are at least two tasks in A (recall that $m > 1$), and no task can precede more than one task in the next timestep. Again, a contradiction.

As a consequence, our assumption that $\tilde{t}_i$ can not be defined, is wrong. We conclude that $\tilde{t}_i$ and t_{i-1} are well defined, for all i. Hence, the sets $\chi_r, \ldots, \chi_1$ are well defined, too. We will show later that each set χ_i is actually nonempty.

Next, we turn to property 1 of Definition 7.7. We observe that $N(t_i) \subseteq N(\tilde{t}_i)$ and $\chi_i \subseteq N(t_i)$. Consequently, $\chi_i = N(t_i) \cap Q(t_{i-1})$, for $i = 1, \ldots, r$. Since $N(t_{top}) = T$ and $t_r = t_{top}$, we obtain $\chi_r = Q(t_{r-1})$. It is furthermore clear that $Q(t_0) = T$. We conclude that $t_0, \ldots, t_{r-1}$ fulfill property 1 of Definition 7.7. As a consequence, every task in χ_i precedes all tasks in χ_{i-1}. To see property 2, consider the following claims.

Claim 1: all tasks in T_{i-1} and all tasks scheduled in timesteps strictly between $\tau_i + 1$ and τ_{i-1} belong to χ_i.

Claim 2: every task in χ_i either belongs to T_{i-1} or is scheduled in a timestep strictly between τ_i and τ_{i-1}.

Claim 3: if $|T_i| = 1$ and $i < r$, then exactly one task of χ_i is mapped to $\tau_i + 1$.

For the time being, assume that all three claims hold. Claim 1 implies that each set χ_i is nonempty, since T_{i-1} contains t_{i-1}. In order to respect communication delays, no task of χ_i is mapped to $\tau_i + 1$ if $|T_i| > 1$, since every task of T_i precedes all tasks of χ_i. By construction, no task of χ_r is mapped to $\tau_r + 1$, and no timestep strictly between $\tau_i + 1$ and τ_{i-1} is partial. Hence,

$$\tau_{i-1} = \begin{cases} \left\lceil \frac{|\chi_i|}{m} \right\rceil & \text{if } i = r, \\ \tau_i + 1 + \left\lceil \frac{|\chi_i| - 1}{m} \right\rceil & \text{if } i < r \text{ and } |T_i| = 1, \\ \tau_i + 1 + \left\lceil \frac{|\chi_i|}{m} \right\rceil & \text{if } i < r \text{ and } |T_i| > 1. \end{cases}$$

Observe that $\chi_r, \ldots, \chi_1$ are scheduled by S the same way as these sets would be scheduled in a UECT packing. Therefore, $|T_i| = 1$ iff $\phi(\chi_r, \ldots, \chi_{i+1}) = 1$, for $i = 0, \ldots, r-1$. We obtain

$$\tau_0 = \left\lceil \frac{|\chi_r|}{m} \right\rceil + \sum_{i=1}^{r-1} \left(1 + \left\lceil \frac{|\chi_i| - \phi(\chi_r, \ldots, \chi_{i+1})}{m} \right\rceil \right) = \mathit{length}(\chi_r, \ldots, \chi_1).$$

Since $\tau_0 = length(S)$ and no schedule for $(T, \prec)$ can have a length shorter than $length(\chi_r, \ldots, \chi_1)$, by Lemma 7.6, we obtain $opt = length(\chi_r, \ldots, \chi_1)$. (As a byproduct, this implies correctness of Theorem 7.1.) What remains is to prove the claims.

Proof of claim 1: Consider the case $\tau_i = \tau_{i-1} - 1$. There is a task scheduled in timestep $\tau_{i-1} - 1$ that precedes all tasks in T_{i-1}. Since $\tilde{t}_i$ has maximal successor set among tasks mapped to timestep τ_i, it holds that $\tilde{t}_i$ precedes all tasks in T_{i-1}. Furthermore, $T_{i-1} \subseteq Q(t_{i-1})$, because t_{i-1} has minimal successor set among tasks in T_{i-1}. Therefore, $T_{i-1} \subseteq \chi_i$.

Otherwise, $\tau_i \leq \tau_{i-1} - 2$. Let x be a task that is either contained in T_{i-1} or mapped to a timestep strictly between $\tau_i + 1$ and τ_{i-1}, such that x has no predecessor mapped to a timestep strictly between $\tau_i + 1$ and τ_{i-1}. In either case, $N(t_{i-1}) \subseteq N(x)$: if x is scheduled in a timestep strictly between $\tau_i + 1$ and τ_{i-1}, this follows from the choice of τ_i; if $x \in T_{i-1}$, this follows because t_{i-1} has minimal successor set among tasks of T_{i-1}.

We prove that x has a predecessor in timestep τ_i. Assume the contrary (i). By construction, either $\tau_i + 1$ is partial or a task u is scheduled in timestep $\tau_i + 1$ with $N(u) \subset N(t_{i-1})$. Hence, if u exists, then x has higher priority than u in L, because L is sorted by nonincreasing successor set and $N(t_{i-1}) \subseteq N(x)$. Since x has no predecessor in timestep τ_i and no predecessor in timesteps strictly between $\tau_i + 1$ and τ_{i-1}, the only possible reason why the list scheduling algorithm left $\tau_i + 1$ partial or mapped u to $\tau_i + 1$ is that a predecessor y of x is mapped to $\tau_i + 1$. Since we assumed that no predecessor of x is mapped to τ_i, it also holds that no predecessor of y is mapped to τ_i. We show next that there exists a task z mapped to $\tau_i + 2$ that has no predecessor in $\tau_i + 1$ and $N(t_{i-1}) \subseteq N(z)$. We have to consider two cases:

case 1: $\tau_i = \tau_{i-1} - 2$. Then $x \in T_{i-1}$, since there are no timesteps strictly between $\tau_i + 1$ and τ_{i-1}. Assume $|T_{i-1}| = 1$. Then $T_{i-1} = \{x\}$, and there exists a task scheduled in timestep $\tau_{i-1} - 1$ that precedes all tasks in T_{i-1}, namely y. We obtain a contradiction, since in this case, τ_i should have been set to $\tau_{i-1} - 1$. Hence, $|T_{i-1}| > 1$. If more than one task in T_{i-1} has a predecessor in timestep $\tau_{i-1} - 1$, then, by definition of interval orders, there would exist a task in timestep $\tau_{i-1} - 1$ that has more than one successor mapped to τ_{i-1}, which can not happen due to communication delays. Hence, there exists a task $z \in T_{i-1}$ that has no predecessor in timestep $\tau_{i-1} - 1$. Note that $N(t_{i-1}) \subseteq N(z)$, since t_{i-1} is a task in T_{i-1} with minimal successor set.

case 2: $\tau_i < \tau_{i-1} - 2$. Then $\tau_i + 2$ is not partial, by the choice of τ_i. We observe that at most one task in $\tau_i + 2$ has a predecessor in $\tau_i + 1$, since otherwise there would exist a task in timestep $\tau_i + 1$ that precedes two tasks in $\tau_i + 2$, by definition of interval orders, and such a situation would violate communication delays. Since $m > 1$, there exists a task z scheduled in timestep $\tau_i + 2$ that has no predecessor in timestep $\tau_i + 1$. Note that $N(t_{i-1}) \subseteq N(z)$, by the choice of τ_i.

Assume there exists a task t in timestep τ_i that is a predecessor of z (ii). Then $t \prec z$ and $y \prec x$. By definition of interval orders, this implies that $t \prec x$ or $y \prec z$. In either case, we obtain a contradiction: if $t \prec x$, then x has a predecessor in timestep τ_i, and if $y \prec z$, then z has a predecessor in timestep $\tau_i + 1$. Hence, assumption (ii) is wrong and z has no predecessor in timestep τ_i. As noted above, z has no predecessor in timestep $\tau_i + 1$ either. It follows that z is available at timestep $\tau_i + 1$. If $\tau_i + 1$ is partial, then the list scheduling algorithm should schedule z in timestep $\tau_i + 1$. If u exists and $\tau_i + 1$ is not partial, then the list scheduling algorithm should schedule z in timestep $\tau_i + 1$ instead of u, since z has higher priority than u in L, because $N(u) \subset N(t_{i-1}) \subseteq N(z)$. A contradiction. Hence, assumption (i) is wrong and x has a predecessor mapped to timestep τ_i.

Let B denote the set of tasks either mapped to some timestep strictly between $\tau_i + 1$ and τ_{i-1} or contained in T_{i-1}. As we have just shown, every task of B has a predecessor either in B or in timestep τ_i. By transitivity of $\prec$, every task of B has a predecessor in timestep τ_i. Let A denote the set of those predecessors. By definition of interval orders, every task in A with maximal successor set precedes all tasks in B. It follows that $\tilde{t}_i$ precedes all tasks in B, because $\tilde{t}_i$ has maximal successor set among tasks mapped to τ_i. Moreover, for every $x \in B$, $N(t_{i-1}) \subseteq N(x)$. We obtain $B \subseteq \chi_i$, which proves claim 1. To prove claim 2 and 3 we require

Claim 4: no task mapped to $\tau_i + 1$ is predecessor of all tasks in χ_i.

Proof of claim 4: Let us first consider the case $\tau_i = \tau_{i-1} - 1$. The tasks of T_{i-1} are mapped to τ_{i-1} and they belong to χ_i, by claim 1. Hence, no task mapped to $\tau_i + 1$ $(= \tau_{i-1})$ is predecessor of all tasks in χ_i. Otherwise, $\tau_i \leq \tau_{i-1} - 2$. Assume x is a task in timestep $\tau_i + 1$ that precedes all tasks of χ_i. Let $\tau_i = \tau_{i-1} - 2$. By claim 1, x is predecessor of all tasks in T_{i-1}. Since x is mapped to timestep $\tau_{i-1} - 1$, τ_i should have been set to $\tau_{i-1} - 1$, a contradiction. Otherwise, $\tau_i < \tau_{i-1} - 2$. Since x precedes the tasks in χ_i, at most one task of χ_i is mapped to $\tau_i + 2$, in order to respect communication delays. By claim 1, all tasks scheduled in timestep $\tau_i + 2$ belong to χ_i. Again a contradiction, since $\tau_i + 2$ is not partial and $m > 1$. Hence, our assumption is wrong and no task in $\tau_i + 1$ precedes all tasks in χ_i.

Proof of claim 2: Assume there exists a task $x \in \chi_i$ mapped to a timestep $\tau > \tau_{i-1}$. Since x precedes all tasks in χ_{i-1}, it is scheduled before any task of χ_{i-1}. The tasks of T_{i-2} are mapped to τ_{i-2} and, by claim 1, all of them belong to χ_{i-1}. Hence, τ is strictly between τ_{i-1} and τ_{i-2}. Therefore, $\tau_{i-1} \leq \tau_{i-2} - 2$. If $\tau_{i-1} < \tau_{i-2} - 2$, then all tasks mapped to $\tau_{i-1} + 2$ are tasks from χ_{i-1} (claim 1), and $\tau_{i-1} + 2$ is not partial, by construction. If $\tau_{i-1} = \tau_{i-2} - 2$, then the tasks of T_{i-2} are mapped to $\tau_{i-1} + 2$. In either case, at least one task of χ_{i-1} is scheduled in timestep $\tau_{i-1} + 2$. Hence, x must be mapped to $\tau_{i-1} + 1$. By claim 4, x is not predecessor of all tasks in χ_{i-1}, a contradiction to the fact that $x \in \chi_i$. Our assumption is therefore wrong, and no task of χ_i is mapped to a timestep $> \tau_{i-1}$. By construction, all tasks mapped to τ_{i-1} that are predecessors of all tasks in χ_{i-1} are part of T_{i-1}. Hence, if a task $x \in \chi_i$ is mapped to τ_{i-1}, then $x \in T_{i-1}$. Since no task of χ_i is scheduled in a timestep earlier than $\tau_i + 1$, we conclude that every task $x \in \chi_i$ is either contained in T_{i-1} or is scheduled in a timestep strictly between τ_i and τ_{i-1}.

Proof of claim 3: Let us first observe that at most one task of χ_i is mapped to $\tau_i + 1$, because there is a task in τ_i, namely $\tilde{t}_i$, that precedes all tasks of χ_i. The claim trivially holds in case $\tau_i = \tau_{i-1} - 1$, because the tasks of T_{i-1} are mapped to τ_{i-1} $(= \tau_i + 1)$, and they belong to χ_i, by claim 1. Otherwise, $\tau_i \leq \tau_{i-1} - 2$. Let $|T_i| = 1$, *i.e.*, $T_i = \{t_i\}$ and $\tilde{t}_i = t_i$. Assume no task of χ_i is mapped to $\tau_i + 1$. Let x be a task in χ_i with minimal predecessor set. Note that every predecessor of x is predecessor of all tasks in χ_i. As a consequence, the only predecessor of x mapped to τ_i is t_i, since otherwise $|T_i| > 1$. Furthermore, no predecessor of x is mapped to $\tau_i + 1$, since otherwise this predecessor would precede all tasks in χ_i, contradicting claim 4. It follows that x is ready at timestep $\tau_i + 1$.

By construction, timestep $\tau_i + 1$ is partial or a task u is mapped to $\tau_i + 1$ with $N(u) \subset N(t_{i-1})$. In the latter case, x has higher priority in L than u, because $x \in \chi_i$ and hence, $N(t_{i-1}) \subseteq N(x)$. In either case, the list scheduling algorithm considered x when it looked for tasks to schedule in timestep $\tau_i + 1$. The only possible reason why the list scheduler did not map x to timestep $\tau_i + 1$ is that there exists a task z mapped to $\tau_i + 1$ such that z and x have a common predecessor in timestep τ_i. This common predecessor must be t_i, since t_i is the only predecessor of x in timestep τ_i. The list scheduling algorithm preferred to schedule z in timestep $\tau_i + 1$ rather than x, hence, z has higher priority than x in L. In other words, $N(x) \subseteq N(z)$. Clearly, $N(t_{i-1}) \subseteq N(x)$, because x belongs to χ_i. As a consequence, $N(t_{i-1}) \subseteq N(z)$ and therefore $z \in \chi_i$ (recall that $\tilde{t}_i = t_i \prec z$). But this contradicts our assumption that no task of χ_i is mapped to $\tau_i + 1$. Hence, our assumption is wrong and some task of χ_i is scheduled in timestep $\tau_i + 1$. □

7.7 Distances and Interfaces

In this section we describe how to efficiently extract information on the structure of an optimal UECT schedule from the interval order, without actually computing the schedule.

Let U be some subset of T. The task system $(U, \prec)$ is still an interval order. Furthermore, $N(x) \subseteq N(y)$ implies $(N(x) \cap U) \subseteq (N(y) \cap U)$, for all $x, y \in U$. As a consequence, any list of all tasks U, ordered by nonincreasing size of their successor sets in $(T, \prec)$, is also ordered by nonincreasing size of their successor sets in $(U, \prec)$. Let L be such a list, and let S be the UECT list schedule for L. We are interested in the length of S, and whether the last timestep of S contains one task or more than one task.

Definition 7.9 *Let $U \subseteq T$. The* scheduling distance *of U, denoted by $D(U)$, is the length of an optimal m-processor UECT schedule for $(U, \prec)$.*

Clearly, the length of S is $D(U)$, since S is an optimal schedule for $(U, \prec)$, by Theorem 7.1. It turns out that the number of tasks mapped to the last timestep of S does not depend on a specific list L, but only on the set U, as long as tasks in L are ordered by nonincreasing successor set. In fact, the number of tasks in the last timestep only depends on the possible block decompositions for $(U, \prec)$.

Definition 7.10 *Let $U \subseteq T$. The* scheduling interface *of U, denoted by $F(U)$, is 0 if there exists an m-processor UECT block decomposition $\chi_r, \ldots, \chi_1$ for $(U, \prec)$ with $\phi(\chi_r, \ldots, \chi_1) = 0$, and it is 1 otherwise.*

In the following we show that the scheduling interface of U is 1 iff the number of tasks in the last timestep of S is 1.

Lemma 7.11 *Let $U \subseteq T$, and let L be a list of all tasks U, ordered by nonincreasing successor set. Let S be the UECT list schedule for L, and let k denote the number of tasks mapped to the last timestep of S. Then $F(U) = 1$ iff $k = 1$.*

Proof. Let $\chi_r, \ldots, \chi_1$ be a UECT block decomposition for S, as constructed in the proof of Theorem 7.8. Recall that in this construction T_0 consists of all tasks mapped to the last timestep of S, and $|T_0| = 1$ iff $\phi(\chi_r, \ldots, \chi_1) = 1$. Let $F(U) = 1$. It follows that $\phi(\chi_r, \ldots, \chi_1) = 1$, and hence, $k = |T_0| = 1$.

Now let $k = 1$. We prove by contradiction that no block decomposition $\chi_r, \ldots, \chi_1$ for $(U, \prec)$ exists with $\phi(\chi_r, \ldots, \chi_1) = 0$. Without loss of generality, we can assume that there exists a task x that is successor of all tasks in U. Since only one task of U is scheduled in the last timestep of S, we can append x to S, and obtain a schedule S' for $(U \cup \{x\}, \prec)$. Since S is an optimal schedule for $(U, \prec)$, by Theorem 7.1, it holds that S' is an optimal schedule for $(U \cup \{x\}, \prec)$. Hence, $D(U \cup \{x\}) = D(U) + 1$. Let $\chi_r, \ldots, \chi_1$ be a UECT block decomposition for $(U, \prec)$ with $\phi(\chi_r, \ldots, \chi_1) = 0$. According to (7.1), a UECT packing of the sets $\chi_r, \ldots, \chi_1, \{x\}$ has length

$$\mathit{length}(\chi_r, \ldots, \chi_1, \{x\}) = \mathit{length}(\chi_r, \ldots, \chi_1) + 1 + \left\lceil \frac{|\{x\}| - \phi(\chi_r, \ldots, \chi_1)}{m} \right\rceil,$$

which equals $D(U) + 2$ because $\mathit{length}(\chi_r, \ldots, \chi_1) = D(U)$, according to Definition 7.7. No UECT schedule for $(U \cup \{x\}, \prec)$ can have a length shorter than $\mathit{length}(\chi_r, \ldots, \chi_1, \{x\})$, by Lemma 7.6, and we obtain $D(U \cup \{x\}) \geq D(U) + 2$. A contradiction. It follows that our assumption is wrong, and no block decomposition $\chi_r, \ldots, \chi_1$ for $(U, \prec)$ exists with $\phi(\chi_r, \ldots, \chi_1) = 0$. Hence, $F(U) = 1$.

□

Our aim is to determine $D(U)$ and $F(U)$ for certain subsets U of T. More precisely, we want to compute $D(P(x))$ and $F(P(x))$ for every $x \in T$. For technical reasons, we compute $D(Q(x))$ and $F(Q(x))$ instead. It is not difficult to compute the former given the latter: For each x, we take a predecessor y of x with minimal successor set. Then $P(x) = Q(y)$.

The data structure used to determine scheduling distances and scheduling interfaces is the *UECT distance graph*. In this graph, there are two edges for every possible block χ_i of some block decomposition for $(Q(x), \prec)$. One edge represents the block in a block decomposition $\chi_r, \ldots, \chi_1$ where $\phi(\chi_r, \ldots, \chi_{i+1}) = 1$, and the other edge represents the same block in a block decomposition with $\phi(\chi_r, \ldots, \chi_{i+1}) = 0$. The weight of an edge equals the number of timesteps required to schedule the block in the respective context. As a consequence, for every task x, and for every UECT block decomposition for the task system $(Q(x), \prec)$, there is a corresponding path in the distance graph and its length is $D(Q(x))$.

Definition 7.12 *Let the m-*processor UECT distance graph *for $(T, \prec)$ be the weighted directed graph defined as follows: Its vertex set consists of a root vertex v, and for every $x \in T$, two vertices $x^{(0)}$ and $x^{(1)}$. Its edge set consists of*

1. *an edge $(v, x^{(b)})$, for every $x \in T$, with $b = \phi(Q(x))$ and*

2. *an edge $(x^{(a)}, y^{(b)})$, for $a \in \{0,1\}$ and every $x, y \in T$ with $|N(x) \cap Q(y)| > 0$, and $b = append(a, N(x) \cap Q(y))$.*

The weight of an edge $(v, x^{(b)})$ equals $\lceil |Q(x)| / m \rceil$, and the weight of an edge $(x^{(a)}, y^{(b)})$ is $1 + \lceil (|N(x) \cap Q(y)| - a) / m \rceil$.

In the rest of this section, let H denote the m-processor UECT distance graph for the interval order $(T, \prec)$. Note that H is acyclic. In order to show that H can be used to determine $D(Q(x))$ and $F(Q(x))$, we need a series of auxiliary propositions. In the first of them, we prove that the length of a path q, starting at the root of H, is the length of the UECT packing of the task sets that correspond to the edges of q. Next, we show that the length of any path ending at $x^{(0)}$ or $x^{(1)}$ does not exceed $D(Q(x))$. Then, we prove that for every task x and every block decomposition of $(Q(x), \prec)$, there exists a path q in H from its root to $x^{(a)}$ such that q has length $D(Q(x))$ and a is the packing interface of the block decomposition. Finally, we show that there is a path from the root of H to $x^{(0)}$ of length $D(Q(x))$ iff $F(Q(x)) = 0$.

Lemma 7.13 *Let $q = v, x_{r-1}^{(a_{r-1})}, \ldots, x_0^{(a_0)}$ be a path in H, starting at the root. Let U_r denote the set $Q(x_{r-1})$, and let U_i denote $N(x_i) \cap Q(x_{i-1})$, for $i = 1, \ldots, r-1$. Then $a_0 = \phi(U_r, \ldots, U_1)$ and the length of q is $length(U_r, \ldots, U_1)$.*

Proof. We claim $a_i = \phi(U_r, \ldots, U_{i+1})$, for $i = 0, \ldots, r-1$. Clearly, the claim holds for $i = r-1$. Let the claim hold for some $i \leq r-1$. Then, $a_{i-1} = append(a_i, U_i)$, by definition of H, and $a_i = \phi(U_r, \ldots, U_{i+1})$, by inductive hypothesis. Since $append(\phi(U_r, \ldots, U_{i+1}), U_i) = \phi(U_r, \ldots, U_i)$, it holds that $a_{i-1} = \phi(U_r, \ldots, U_i)$, which proves the claim by induction on i. By construction of H, the length of q is

$$\left\lceil \frac{|U_r|}{m} \right\rceil + \sum_{i=1}^{r-1} \left(1 + \left\lceil \frac{|U_i| - a_i}{m} \right\rceil \right).$$

We replace a_i with $\phi(U_r, \ldots, U_{i+1})$ and obtain the desired result. □

Lemma 7.14 *Let $x \in T$, and let w be the length of a path from the root of H to $x^{(0)}$ or $x^{(1)}$. Then $D(Q(x)) \geq w$.*

Proof. Let $v, x_{r-1}^{(a_{r-1})}, \ldots, x_0^{(a_0)}$ be a path in H of length w, starting at the root, with $x_0 = x$. Furthermore, let $U_r = Q(x_{r-1})$ and let $U_i = N(x_i) \cap Q(x_{i-1})$, for $i = 1, \ldots, r-1$. By Lemma 7.13, $w = length(U_r, \ldots, U_1)$. It is not difficult to see that $U_r, \ldots, U_1$ are pairwise disjoint nonempty subsets of $Q(x)$, and every task in U_i precedes all tasks in U_{i-1}, for $i = 2, \ldots, r$. Hence, we can apply

Lemma 7.6 and obtain $D(Q(x)) \geq \mathit{length}(U_r,\dots,U_1)$. It follows that $D(Q(x)) \geq w$. □

Lemma 7.15 *Let $x \in T$, and let $\chi_r,\dots,\chi_1$ be an m-processor UECT block decomposition for $(Q(x),\prec)$. Then there exists a path in H from its root to $x^{(a)}$ of length $D(Q(x))$ with $a = \phi(\chi_r,\dots,\chi_1)$.*

Proof. By Definition 7.7, there exist tasks $t_0,\dots,t_{r-1}$ such that $\chi_r = Q(t_{r-1})$, $\chi_i = N(t_i) \cap Q(t_{i-1})$, for $i = 1,\dots,r-1$, and $Q(t_0) = Q(x)$. Hence, $\chi_1 = N(t_1) \cap Q(x)$, and there exists a path $v, t_{r-1}^{(a_{r-1})},\dots,t_1^{(a_1)}, x^{(a)}$ in H, starting at the root. By Lemma 7.13, $a = \phi(\chi_r,\dots,\chi_1)$ and the length of this path is $\mathit{length}(\chi_r,\dots,\chi_1)$, which equals $D(Q(x))$, according to Definition 7.7. □

Lemma 7.16 *Let $x \in T$. There exists a path in H from its root to $x^{(0)}$ of length $D(Q(x))$ iff $F(Q(x)) = 0$.*

Proof. Let $F(Q(x)) = 0$. Then there exists a UECT block decomposition χ_r, $\dots$, χ_1 for $(Q(x),\prec)$ with $\phi(\chi_r,\dots,\chi_1) = 0$. By Lemma 7.15, there exists a path of length $D(Q(x))$ from the root of H to $x^{(0)}$.

Conversely, let $q = v, x_{r-1}^{(a_{r-1})},\dots,x_1^{(a_1)}, x^{(0)}$ be a path of length $D(Q(x))$ from the root of H to $x^{(0)}$. Let U_r denote the set $Q(x_{r-1})$, let $U_i = N(x_i) \cap Q(x_{i-1})$, for $i = 2,\dots,r-1$, and let U_1 denote the set $N(x_1) \cap Q(x)$. By Lemma 7.13, it holds that $\phi(U_r,\dots,U_1) = 0$ and the length of q equals $\mathit{length}(U_r,\dots,U_1)$. As a consequence, $U_r,\dots,U_1$ is a UECT block decomposition for $(Q(x),\prec)$. We conclude that $F(Q(x)) = 0$ since the packing interface of $U_r,\dots,U_1$ is 0. □

We have now everything in hand for an efficient parallel algorithm that computes scheduling distances and scheduling interfaces. As we will show, it suffices to compute longest paths in the distance graph.

Lemma 7.17 *Let $|T| = n$. We can compute $D(P(x))$ and $F(P(x))$ for all tasks $x \in T$, with $P(x) \neq \emptyset$, in time $O(\log^2 n)$ using $n^3/\log n$ EREW PRAM processors or in time $O(\log n)$ using n^3 CRCW PRAM processors.*

Proof. We assume that $(T,\prec)$ is given by a precedence graph G. Let G be represented by an adjacency matrix A such that $a_{ij} = 1$ if there is an edge from task i to task j, and $a_{ij} = 0$ otherwise. If not transitively closed, we compute the transitive closure of G using standard techniques (cf. Subsection 3.12.2).

In the first step, we determine $|Q(x)|$ and $|N(x) \cap Q(y)|$, for all $x,y \in T$. The sets $N(x)$ can directly be obtained from the adjacency matrix of the transitively closed G. The set $Q(y)$ consists of all tasks x with $|N(x)| \geq |N(y)|$. Therefore, $Q(y)$ can be computed using prefix operations. Given $N(x)$ and $Q(y)$, we can determine $N(x) \cap Q(y)$ in constant time using n processors. In order to run without concurrent read, we make a sufficient number of copies of $N(x)$ and $Q(y)$ beforehand using prefix operations. We conclude that $|Q(x)|$ and $|N(x) \cap Q(y)|$ can be computed for all pairs of tasks in time $O(\log n)$ using $n^3 / \log n$ EREW PRAM processors.

Next, we construct the m-processor UECT distance graph H of $(T, \prec)$. For every $x \in T$, we fix some task z that is predecessor of x and has minimal successor set. Clearly, $Q(z) = P(x)$. By Lemma 7.14, any path from the root of H to $z^{(0)}$ or $z^{(1)}$ has length at most $D(Q(z))$. By Lemma 7.16, there is a path of length $D(Q(z))$ from the root of H to $z^{(0)}$ iff $F(Q(z)) = 0$. By Theorem 7.8, there exists at least one m-processor UECT block decomposition $\chi_r, \ldots, \chi_1$ for $(Q(z), \prec)$. If $F(Q(z)) = 1$, then $\phi(\chi_r, \ldots, \chi_1) = 1$, and there exists a path of length $D(Q(z))$ from the root of H to $z^{(1)}$, by Lemma 7.15. Hence, the scheduling distance and the scheduling interface of $P(x)$ can be determined as follows. Let q_0 be a longest path from the root of H to $z^{(0)}$ and let q_1 be a longest path from the root to $z^{(1)}$. At least one of them exists and the length of the longer of them equals $D(Q(z))$. Furthermore, $F(Q(z)) = 1$ iff either q_0 does not exist or its length is shorter than that of q_1.

Therefore, it suffices to determine longest paths in H. By Theorem 3.26, this takes $O(\log^2 n)$ time and $n^3 / \log n$ processors on the EREW PRAM. As for the CRCW PRAM implementation, we observe the following. An optimal UECT schedule for a task system with n tasks has length at most n. Hence, no path in H is longer than n (cf. Lemma 7.14). In this case, the lengths of all longest paths can be obtained on the CRCW PRAM in time $O(\log n)$ using n^3 processors, by Theorem 3.27. □

7.8 Constructing an Optimal Schedule

Using the scheduling distances and the scheduling interfaces of the sets $P(x)$, we construct an instance of a different scheduling problem where tasks are not constrained by precedence but have individual release times and deadlines. An instance of this problem consists of a task system (T, r, d) and a number m of target processors, where r and d are mappings from T to positive integer timesteps. An m-processor schedule for (T, r, d) is a mapping S from T to positive integer timesteps such that

1. $r(x) \leq S(x)$, for all $x \in T$, and

2. no more than m tasks are mapped to the same timestep.

If there exists a schedule for (T,r,d) that meets the deadlines (*i.e.*, $S(x) \leq d(x)$ for all $x \in T$), then the following list scheduling algorithm finds such a schedule [Jac55]. Take a list of all tasks ordered by nondecreasing deadline. Consider one timestep after the other, starting with timestep 1. To find the tasks for timestep τ, scan the list from left and pick, of the tasks with release time $\leq \tau$, as many as possible, up to a maximum of m. Map these tasks to timestep τ and remove them from the list. If the list is not empty by now, repeat the above process for timestep $\tau + 1$.

The schedule obtained by this algorithm is called earliest deadline schedule or ED schedule (cf. Section 3.13). In this section we show that we can find, for each task x, a release time $r(x)$ and a deadline $d(x)$ such that every m-processor ED schedule for (T,r,d) is an optimal m-processor UECT schedule for $(T,\prec)$.

Lemma 7.18 *Let L be a list of all tasks, ordered by nonincreasing successor set, and let S be the UECT list schedule for L. Let τ be the latest timestep in S with a predecessor of x, and let k denote the number of predecessors of x that S schedules in timestep τ. Then $\tau = D(P(x))$ and furthermore, $F(P(x)) = 1$ iff $k = 1$.*

Proof. Let L' be the prefix of L that consists of $P(x)$, and let S' be the UECT list schedule for L'. By Theorem 7.1, S' is an optimal UECT schedule for $(P(x),\prec$), since the tasks in L' are ordered by nonincreasing size of their successor sets in $(P(x),\prec)$. The length of S' is therefore $D(P(x))$. If we restrict S to the tasks in $P(x)$, then S equals S', because L' is a prefix of L. Hence, the last timestep in S with a predecessor of x is timestep $D(P(x))$. By Lemma 7.11, the last timestep of S' contains one task iff $F(P(x)) = 1$. Therefore, S schedules only one predecessor of x in timestep τ iff $F(P(x)) = 1$. □

S schedules task x in no timestep earlier than $D(P(x)) + 1$, since at least $D(P(x))$ timesteps are required to schedule all predecessors of x. If $F(P(x)) = 0$, then at least two predecessors of x are mapped to timestep $D(P(x))$, by Lemma 7.18. In this case, x can not be scheduled in a timestep earlier than $D(P(x)) + 2$, in order to allow for communication.

Definition 7.19 *Let τ be a timestep, let L be a list of all tasks, ordered by nonincreasing successor set, and let x be a task with $P(x) \neq \emptyset$. Then x is* favored task candidate *of τ if $D(P(x)) = \tau$ and $F(P(x)) = 1$. Let $ft(\tau)$ denote the leftmost favored task candidate of τ in L. We call $ft(\tau)$* favored task *of timestep τ in L.*

By Lemma 7.18, exactly one predecessor of every favored task candidate of τ is scheduled by S in timestep τ. It follows that all favored task candidates of timestep τ are ready at timestep $\tau+1$. Furthermore, by definition of interval orders, there exists a task mapped to timestep τ that is predecessor of all favored task candidates of timestep τ. As a consequence, at most one favored task candidate can be mapped to timestep $\tau+1$, in order to respect communication delays. If the list scheduling algorithm schedules a favored task candidate of τ in timestep $\tau+1$ at all, then it will choose one with highest priority in L, *i.e.*, it will choose the favored task of timestep τ.

Lemma 7.20 *Let L be a list of all tasks, ordered by nonincreasing successor set, let S be the UECT list schedule for L, and let $x,y \in T$. If $S(y) = S(x)+1$ and $x \prec y$, then y is the favored task of timestep $S(x)$ in L.*

Proof. By Lemma 7.18, $S(x) = D(P(y))$. Assume that y is not a favored task candidate of timestep $S(x)$. Then $F(P(y)) = 0$. By Lemma 7.18, at least two predecessors of y are mapped to $S(x)$. Hence, y can not be scheduled in a timestep earlier than $S(x)+2$, in order to allow for communication. A contradiction. Therefore, y is favored task candidate of $S(x)$. As already noted, only one favored task candidate of $S(x)$ can be mapped to $S(x)+1$, and all favored task candidates of $S(x)$ are ready at timestep $S(x)+1$. Because the favored task of $S(x)$ has higher priority in L than all other favored task candidates of $S(x)$, it follows that y must be the favored task of $S(x)$ in L. □

In the new task system (T,r,d), the release time of task x will be 1 if x has no predecessors, it will be $D(P(x))+1$ if x is the favored task of timestep $D(P(x))$, and it will otherwise be $D(P(x))+2$. We define deadlines in such a way that the ED scheduling algorithm considers the tasks in the same order as the UECT list scheduling algorithm does. For this purpose, the deadline of task x is set to $|T|$ plus the position of x in L. These deadlines are large enough that there exists a schedule for (T,r,d) that meets the deadlines.

Lemma 7.21 *Let L be a list of all tasks, ordered by nonincreasing successor set, and let (T,r,d) be the task system with release times and deadlines defined as follows. For every $x \in T$, let*

$$\begin{aligned} r(x) &:= \begin{cases} 1 & \text{if } P(x) = \emptyset, \\ D(P(x))+1 & \text{if } x = ft(D(P(x))), \\ D(P(x))+2 & \text{otherwise,} \end{cases} \\ d(x) &:= |T| + \text{the position of } x \text{ in } L. \end{aligned}$$

Then every m-processor ED schedule for (T,r,d) is an optimal m-processor UECT schedule for $(T,\prec)$.

Proof. Let S' be an ED schedule for (T,r,d), and let S be the UECT list schedule for L. We claim that S and S' are identical. Assume the contrary and let τ be the earliest timestep where S and S' differ. Let S'_τ denote the set of tasks that S' schedules in timestep τ, and let S_τ denote the set of tasks that S schedules in timestep τ. Finally, let x be the leftmost task in L that is contained either in $S_\tau - S'_\tau$ or in $S'_\tau - S_\tau$.

Since both scheduling algorithms consider tasks in the order given by L, and x is the leftmost task in L that is handled differently by the ED scheduler and by the UECT list scheduler, we can derive that both algorithms consider x when they look for tasks to be scheduled in timestep τ. As a consequence, if $x \in S_\tau - S'_\tau$, then the only possible reason why the ED scheduler did not map x to timestep τ is that the release time of x is later than τ. Conversely, if $x \in S'_\tau - S_\tau$, then the only two possible reasons why the UECT list scheduler did not map x to timestep τ are that either x is not ready at timestep τ or S already mapped a task y to timestep τ such that x and y have a common predecessor in timestep $\tau - 1$.

We first analyze the case $x \in S'_\tau - S_\tau$, *i.e.*, the UECT list scheduling algorithm did not schedule x in timestep τ, but the ED scheduler did. It follows that $r(x) \leq \tau$. If $P(x) = \emptyset$, then $r(x) = 1$. If x is favored task of $D(P(x))$, then its release time is $D(P(x)) + 1$ and only one predecessor of x is mapped to timestep $D(P(x))$, by Lemma 7.18. Otherwise, the release time of x is $D(P(x)) + 2$. By Lemma 7.18, the latest timestep in S with a predecessor of x is $D(P(x))$. In any case, x is ready in S at timestep τ, since $r(x) \leq \tau$. As a consequence, the only possible reason why the UECT list scheduler did not map x to timestep τ is that S already mapped a task y to τ such that x and y have a common predecessor in timestep $\tau - 1$. By Lemma 7.20, y is favored task of timestep $\tau - 1$ in L, hence, x is not favored task of timestep $\tau - 1$. Since x has a predecessor in $\tau - 1$ and is ready at timestep τ, we obtain $D(P(x)) = \tau - 1$. Hence, x is not favored task of timestep $D(P(x))$. Its release time is therefore $D(P(x)) + 2$, which equals $\tau + 1$. Obviously, the ED scheduler can not schedule x in timestep τ. A contradiction.

Now let $x \in S_\tau - S'_\tau$. It follows that the release time of x is greater than τ. The UECT list scheduler does not map x to a timestep earlier than $D(P(x)) + 1$. Hence, $\tau \geq D(P(x)) + 1$. On the other hand, the release time of x is $\leq D(P(x)) + 2$, by definition. It follows that $r(x) = D(P(x)) + 2$ and $\tau = D(P(x)) + 1$. According to Lemma 7.18, timestep $D(P(x))$ contains a predecessor of x.

Hence, x is favored task of timestep $D(P(x))$ in L, by Lemma 7.20. Therefore, its release time must be $D(P(x))+1$, by definition. A contradiction.

We conclude that no such task x exists and therefore $S_\tau = S'_\tau$. It follows that S and S' are identical. According to Theorem 7.1, S is an optimal UECT schedule for $(T,\prec)$. Hence, the same holds for S'. □

We are now able to state the main result of this chapter.

Theorem 7.22 *Let $(T,\prec)$ be an interval order, with $|T| = n$. An optimal m-processor UECT schedule for $(T,\prec)$ can be computed in time $O(\log^2 n)$ using $n^3/\log n$ EREW PRAM processors or in time $O(\log n)$ using n^3 CRCW PRAM processors.*

Proof. According to Lemma 7.17, we can compute the scheduling distance and the scheduling interface of $P(x)$, for all tasks x simultaneously, within the desired resource bounds. Given these numbers, it is easy to construct the task system with release times and deadlines defined in Lemma 7.21, using sorting and prefix operations (Theorems 3.1 and 3.9). Parallel algorithms that compute ED schedules have been discussed in Section 3.13. The algorithm of Dekel and Sahni runs in time $O(\log^2 n)$ using $n/\log n$ processors, while the algorithm of Dolev, Upfal, and Warmuth requires only $O(\log n)$ time but $n^2/\log n$ processors. Both algorithms run on the EREW PRAM. □

What remains is to assign tasks to processors. Let $x_1 \prec x_2 \prec \cdots \prec x_s$, $s > 1$, be tasks scheduled in successive timesteps, *i.e.*, $S(x_i) = S(x_{i-1})+1$, for $i = 2,\ldots,s$, such that no predecessor of x_1 is scheduled in timestep $S(x_1)-1$ and no successor of x_s is scheduled in timestep $S(x_s)+1$. Call any such set of tasks a *chain* in S. Clearly, all tasks in a chain must be assigned to the same processor. In UECT schedules for interval orders, at most one task scheduled in timestep τ has a predecessor in $\tau-1$, and at most one task in timestep τ has a successor in $\tau+1$. As a consequence, chains do not overlap in S by more than one timestep. For instance, no task of any chain other than $x_1,\ldots,x_s$ is scheduled in timesteps $S(x_2),\ldots,S(x_{s-1})$. We order the chains of S by the timestep they start and number them. All tasks in a chain with an odd number are assigned to processor 1 and all tasks in a chain with an even number are assigned to processor 2. Next, for each timestep, we number the tasks that are not part of a chain, starting with number 1 and skipping number 1 and/or number 2, depending on which processors are already used by chains in the respective timestep. Finally, we assign all tasks with number i to processor i. All of these operations can be done using pointer jumping and prefix operations within the resource bounds given in Theorem 7.22.

CHAPTER 8

Closing Remarks

In the last few chapters, we have dealt with fundamental scheduling problems relevant to parallel and networked computing. These problems have in common that their sequential solutions are based on relatively “simple” strategies. However, obtaining fast and work efficient parallel algorithms for them, turned out to be much more difficult.

The first problem that we have analyzed is the multiprocessor scheduling problem with unit processing times and tree precedence constraints. Hereby, one focus is on the influence of the number m of target processors on the complexity of the scheduling problem. We have presented an EREW PRAM algorithm that runs in time $O(\log n \log m)$ if $n / \log n$ processors are available. This shows that if m is fixed and not part of the input, then $O(\log n)$ time and $O(n)$ operations suffice to determine an optimal schedule. But even if m is part of the problem instance, our algorithm still represents an improvement on previous work. Compared with the fast $O(\log n)$ time algorithm in [DUW86], the number of operations is considerably reduced with only a moderate increase in running time. Compared with the $O(\log^2 n)$ time algorithm in [DUW86], both time and work are reduced.

Some questions remain open in this context. As far as the number of operations is concerned, our algorithm is fairly close to optimality, but there is still a gap when compared with the linear time sequential algorithm given in [BGJ77]. Whether this gap can be closed, is an intriguing open problem. Second, it is worth noting that the schedules computed by our algorithm do not conform to the highest level first strategy. It remains an open problem whether HLF schedules can be computed using the same amount of resources.

The next problem that attracted our attention is the two processor scheduling problem. Given a set of n tasks with unit processing times, constrained by an arbitrary precedence relation, the problem is to find a two processor schedule for these tasks of minimal length. We have presented an algorithm that solves this problem on the CREW PRAM in time $O(\log^2 n)$ using $n^3/\log n$ processors. Compared to the best previous algorithm, which requires n^5 processors, our algorithm represents a major improvement.

An interesting application of our two processor scheduling algorithm is the maximum matching problem in co-comparability graphs (such as interval graphs and permutation graphs). By combining our result with the transitive orientation algorithm of Morvan and Viennot [MV96], we have shown that maximum matchings in complements of comparability graphs can be computed in $O(\log^2 n)$ time if n^3 CREW PRAM processors are used. This represents a significant improvement on previously known bounds, since the previous algorithms for this problem are also based on a solution for the two processor scheduling problem, and therefore require n^5 processors.

The next problem that we have considered is closely related to the two processor scheduling problem. We have shown that if the precedence relation is restricted to be a series parallel order, then an optimal two processor schedule can be computed much more efficiently than in the general case. We have presented an algorithm that computes optimal two processor schedules for series parallel orders in $O(\log n)$ time on $n/\log n$ EREW PRAM processors, provided that a decomposition tree for the precedence graph is given. In those cases where the decomposition tree is not given, our algorithm requires time $O(\log(n+e)+\log n \log^* n)$ on the EREW PRAM and time $O(\log(n+e))$ on the CRCW PRAM, where e is the number of edges in the precedence graph. In either case, the number of operations performed is $O(n+e)$, which is asymptotically optimal.

Finally, we have turned to the problem of scheduling with communication delays. First, we have shown that the problem of computing optimal m-processor schedules for interval ordered tasks of unit length, subject to unit delays for interprocessor communication, can be solved by a sequential algorithm in linear time. This improves on results given in [AER95]. Our main contribution, however, is a fast parallel algorithm for this problem. If implemented on an EREW PRAM, our algorithm runs in time $O(\log^2 n)$ using $n^3/\log n$ processors. If implemented on a CRCW PRAM, it requires only $O(\log n)$ time and uses n^3 processors. Our algorithm is the first $\mathcal{NC}$ algorithm for this problem.

As far as the number of operations is concerned, our parallel algorithm is of course not optimal. It is an interesting question whether one can find a more efficient $\mathcal{NC}$ algorithm, assuming even that the input precedence graph is already transitively closed or the interval order is given by an interval representation.

8.1 Future Work

Our research on parallel scheduling algorithms can be extended and continued in several directions. We illustrate some of these directions in the following.

A promising area for investigations is scheduling with release times and deadlines. It is often the case that execution of tasks is not only constrained by precedence but also by timing constraints imposed by the environment. For instance, a task may require data from an outside agent or a resource required to complete a task becomes unavailable after a certain point in time. In these cases, execution of a task can not start before a given release time r_i and/or must be finished before a given deadline d_i.

We first consider the case where no precedence constraints are present. If tasks have unit length and individual integer release times and deadlines, then, as we have seen in Section 3.13, it is not difficult to compute a schedule that meets the deadlines if such a schedule exists at all. The difficulty of this problem radically changes if tasks have individual processing times or the release times and deadlines are nonintegral. If the processing times of tasks are not equal, then the problem of determining whether a schedule exists that meets all deadlines, is $\mathcal{NP}$-complete, even for the single processor case [GJ77]. If the release times and deadlines are nonintegral and tasks have unit length, then polynomial solutions exist, but they are much more involved than in the case with integer times. The main difficulty that one has to cope with is the existence of "forbidden regions" in optimal schedules. At some given point in time it might be inappropriate to start the execution of an available task, instead it is necessary to leave a processor idle and to wait until another task's release time is reached. Moreover, the decisions to leave processors idle for certain periods of time have great impact on the global structure of the schedule, making it difficult to base these decisions on a local strategy. Early work on this subject was done by Simons [Sim78, Sim80]. Subsequently, improved algorithms have been given in [GJST81, SW89]. The fastest algorithm for the single processor case (multiprocessor case) runs in time $O(n \log n)$ ($O(mn^2)$). An $\mathcal{NC}$ algorithm for the single processor case has recently been developed by

Frederickson and Rodger [FR94]. It runs in time $O(\log^2 n)$ and uses $n^4 / \log n$ CREW PRAM processors. An interesting open problem is to determine the parallel complexity of the multiprocessor case.

Let us now turn to the case where precedence constraints are present. It is assumed that all tasks have unit processing time and all release times and deadlines are integers. The basic idea that leads to algorithms for scheduling with precedence constraints *and* deadlines is "deadline modification" [GJ76]. If t_i is a task and T is the set of successors of t_i with deadlines $\leq d$, then t_i must be finished by time $d - \lceil |T| / m \rceil$ in any schedule that meets the deadlines. Hence, we can savely set the deadline of t_i to the minimum of its old deadline and $d - \lceil |T| / m \rceil$. By repeating these deadline modifications for all tasks and all possible deadlines until no more changes occur, one obtains deadlines that are "consistent" with the precedence constraints. If we put all tasks into a list sorted by nondecreasing deadline, then, under certain assumptions concerning the precedence constraints or the number of processors m, the list schedule obtained from this list meets the deadlines. If, in addition, tasks have individual release times, the deadline modification becomes more involved but the basic idea remains the same [GJ77].

Brucker, Garey, and Johnson have obtained interesting results concerning the complexity of scheduling trees with deadlines. For the case of intrees without release times, they have given an $O(n \log n)$ algorithm that minimizes the maximum number of timesteps any task is scheduled behind its deadline. In the case of outtrees, the problem of deciding whether a schedule exists that meets the deadlines, is $\mathcal{NP}$-hard [BGJ77]. Also remarkable is that the two processor scheduling problem is still solvable in polynomial time if tasks have individual release times and deadlines. A sequential algorithm that runs in time $O(n^3)$ has been given by Garey and Johnson [GJ77]. For the case, where all release times are equal (*i.e.*, only deadlines are present), Helmbold has developed an $\mathcal{NC}$ algorithm [Hel86]. The number of required processors is quite large and it is a challenging problem to find a more efficient $\mathcal{NC}$ algorithm, possibly by utilizing some of the techniques introduced in Chapter 5. It turns out that the ideas of Garey and Johnson used in the two processor algorithm can also be applied to the problem of scheduling interval ordered tasks on a variable number of processors. Krauß has given an $\mathcal{NC}$ algorithm that computes m-processor schedules for interval orders with release times and deadlines [Kra93]. The algorithm requires n^8 processors and we conjecture that a more efficient parallel algorithm can be found.

Recently, Verriet has generalized the ideas of Garey and Johnson to scheduling with communication delays. In order to account for communication, his algorithm computes modified deadlines for *pairs* of tasks. In [Ver96], he has given an algorithm that optimally schedules interval ordered tasks with release times and deadlines, subject to unit communication delays. It would be interesting to find a fast parallel solution to this problem.

Another challenging area for research is the development of parallel algorithms that compute near-optimal schedules. Most scheduling problems are $\mathcal{NP}$-complete if we wish to find optimal solutions. In some applications, however, it is sufficient to find schedules that are (provably) quite close to the optimum. Sequential approximation algorithms for $\mathcal{NP}$-hard scheduling problems have received considerable attention in the past ([Gra69, CGJ78, HS85, RS87, HM95, MSS96, VRKL96], to name only a few). As far as parallel approximation algorithms for scheduling are concerned, not much is known. Early work on this subject can be found in [DUW86] and [May88]. In both papers, parallel $(2-\frac{1}{m})$-approximation algorithms for scheduling unit processing time tasks with arbitrary precedence constraints can be found. In [May85], an $\mathcal{NC}$ approximation scheme for scheduling independent tasks has been given (cf. [May88]). It is essentially a parallel version of the algorithm of Hochbaum and Shmoys [HS85], and is based on discretization. The idea underlying this approach is to round off the task lengths to multiples of a very small value that depends on the desired quality of the schedule. After rounding, only a few different task lengths remain, and an optimal solution for these modified tasks can be determined using dynamic programming. In the end, the original task lengths are restored, thus obtaining a schedule that exceeds the optimum by a small ε-fraction only. The actual algorithm is somewhat more complicated, *e.g.*, one has to take care of tasks that are too small for rounding.

A key role in many sequential approximation algorithms plays linear programming. In general, solving linear programs is $\mathcal{P}$-complete [DLR79]. It is also $\mathcal{P}$-complete to find approximate solutions for linear programs [Ser91]. Hence, it is highly unlikely that a fast parallel algorithm for general LPs can be found. One way to work around this deficiency is to find subclasses of linear programs that can be solved in $\mathcal{NC}$ but are still useful to approximate scheduling. In [LN93], Luby and Nisan have given an $\mathcal{NC}$ approximation scheme for LPs where all coefficients are nonnegative. Such LPs are called *positive linear programs* (PLP). Although positive linear programs seem to be quite restricted, the algorithm of Luby and Nisan has been successfully applied to combinatorial optimization problems, *e.g.*, MAX SAT and MAX DIRECTED CUT [Tre96]. Recently, Serna and Xhafa have applied the algorithm of Luby and Nisan to the problem of scheduling with unrelated processors. In this problem, tasks are independent and for each task t_i and each processor P_j, a processing time p_{ij} is given, expressing that t_i requires time p_{ij} if executed by processor P_j. The goal is to find a schedule of minimal length. For the case where the number of processors m is fixed and not part of the input, a parallel $(2+\varepsilon)$-approximation algorithm is given in [SX97]. The approach taken in [SX97] looks promising and it is a challenging open problem to find other parallel approximation algorithms for scheduling problems using $\mathcal{NC}$ approximation algorithms for restricted linear programs.

Bibliography

[ADKP89] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 10:287–302, 1989.

[AER95] H. H. Ali and H. El-Rewini. An optimal algorithm for scheduling interval ordered tasks with communication on *n* processors. *J. Comput. Syst. Sci.*, 51(2):301–306, 1995.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput.*, 16(5):808–835, October 1987.

[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, 1988. Springer-Verlag.

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

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BdF96] H. L. Bodlaender and B. de Fluiter. Parallel algorithms for series parallel graphs. In J. Diaz and M. Serna, editors, *Proceedings of the 4th Annual European Symposium on Algorithms, ESA'96 (Barcelona, Spain, September 1996)*, LNCS 1136, pages 277–289, Berlin, 1996. Springer-Verlag.

[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.

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, April 1989.

[BOV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree form. *ACM Trans. Program. Lang. Syst.*, 7(2):348–357, April 1985.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[Bru82] J. 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] C. C.-Y. Chen and S. 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, Los Alamitos-Washington-Brussels-Tokyo, 1991. IEEE Computer Society Press.

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

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97, February 1986.

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

[CGJ78] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17, February 1978.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, 1988.

[CV88a] 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.

[CV88b] R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and 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 91–100, Berlin, 1988. Springer-Verlag.

[DLR79] D. Dobkin, R. J. Lipton, and S. Reiss. Linear programming is log-space hard for $\mathcal{P}$. *Inf. Process. Lett.*, 8(2):96–97, February 1979.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10(4):657–675, November 1981.

[DS83] E. Dekel and S. Sahni. Parallel generation of postfix and tree forms. *ACM Trans. Program. Lang. Syst.*, 5(3):300–317, July 1983.

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

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

[Edm65] J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

[Epp92] D. Eppstein. Parallel recognition of series-parallel graphs. *Inf. Comput.*, 98(1):41–55, May 1992.

[FG65] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15:835–855, 1965.

[Fis85] P. C. Fishburn. *Interval Orders and Interval Graphs.* John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1985.

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

[FLMB96] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET-UCT series-parallel graphs on two processors. *Theoretical Computer Science*, 162:323–340, 1996.

[FR94] G. N. Frederickson and S. H. Rodger. An $\mathcal{NC}$ algorithm for scheduling unit-time jobs with arbitrary release times and deadlines. *SIAM J. Comput.*, 23(1):185–211, February 1994.

[FRW88] F. E. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, 1988.

[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 Press.

[Gab81] H. N. Gabow. A linear-time recognition algorithm for interval DAGS. *Inf. Process. Lett.*, 12(1):20–22, February 1981.

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

[Gav72] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1:180–187, 1972.

[GH64] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Can. J. Math.*, 16:539–548, 1964.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[GJ76] M. R. Garey and D. S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *J. ACM*, 23(3):461–467, July 1976.

[GJ77] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6(3):416–426, September 1977.

[GJ78] M. R. Garey and D. S. Johnson. “Strong” $\mathcal{NP}$-completeness results: Motivation, examples, and implications. *J. ACM*, 25:499–508, 1978.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, New York-San Francisco, 1979.

[GJST81] M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM J. Comput.*, 10:256–269, 1981.

[Glo67] F. Glover. Maximum matching in a convex bipartite graph. *Naval Research Logistics Quarterly*, 14:313–316, 1967.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York-San Francisco-London-San Diego, 1980. Computer Science and Applied Mathematics.

[GP88] Z. Galil and V. Pan. Improved processor bounds for combinatorial problems in $\mathcal{RNC}$. *Combinatorica*, 8:189–200, 1988.

[GR89] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Inf. Comput.*, 81(1):32–45, 1989.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, March 1969.

[GT85] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, April 1985.

[GT93] F. Guinand and D. Trystram. Optimal scheduling of UECT trees on two processors. Technical Report APACHE 3, Laboratoire de Modèlisation et de Calcul - IMAG, Grenoble, November 1993.

[Hel86] D. P. Helmbold. *Parallel Scheduling Algorithms.* PhD thesis, Department of Computer Science, Stanford University, 1986.

[HH94] S.-Y. Hsieh and C.-W. Ho. An efficient parallel strategy for recognizing series-parallel graphs. In D.-Z. Du and X.-S. Zhang, editors, *Proceedings of the 5th International Symposium on Algorithms and Computation, ISAAC'94 (Beijing, P. R. China, August 1994)*, LNCS 834, pages 496–504, Berlin, 1994. Springer-Verlag.

[HLV92] J. A. Hoogeveen, J. K. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. Technical Report BS-R9229, CWI, Amsterdam, 1992.

[HM86] D. Helmbold and E. W. Mayr. Perfect graphs and parallel algorithms. In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 853–860, Washington, DC, August 1986. IEEE Computer Society Press.

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

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

[HM95] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on *m* processors with small communication delays. Technical report, Laboratoire LITP, Institut Blaise Pascal, Université Paris VI, 1995.

[HM97] C. Hanen and A. Munier. Using duplication for scheduling unitary tasks on *m* processors with unit communication delays. *Theoretical Computer Science*, 178:119–127, 1997.

[HN60] F. Harary and R. Norman. Some properties of line digraphs. *Rendiconti del Circolo Mathematico Palermo*, 9:149–163, 1960.

[HR89] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Inf. Process. Lett.*, 33:181–185, 1989.

[HS85] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. In *Proceedings of the 26th Ann. IEEE Symposium on Foundations of Computer Science (Portland, OR)*, pages 79–89. IEEE, 1985.

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

[HY87] X. He and Y. Yesha. Parallel recognition and decomposition of two terminal series parallel graphs. *Inf. Comput.*, 75(1):15–38, 1987.

[Jac55] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[JSS91] H. Jung, M. Serna, and P. Spirakis. A parallel algorithm for two processors precedence constraint scheduling. In J. Leach Albert, B. Monien, and M. R. Artalejo, editors, *Proceedings of the 18th International Colloquium on Automata, Languages and Programming (Madrid, Spain, July 1991),* LNCS 510, pages 417–428, Berlin, 1991. Springer-Verlag.

[Jun92] N. Jung. Ein effizienter paralleler Algorithmus für Zweiprozessorscheduling. Master's thesis, Institut für Informatik, Universität Stuttgart, Juli 1992.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of the Workshop on Complexity of Computations (Yorktown Heights),* pages 85–103, New York-London, 1972. Plenum Press.

[Kar86] H. J. Karloff. A Las Vegas $\mathcal{RNC}$ algorithm for maximum matching. *Combinatorica,* 6(4):387–392, 1986.

[KD88] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. 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 101–110, Berlin, 1988. Springer-Verlag.

[Kle93] P. N. Klein. Parallel algorithms for chordal graphs. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 8, pages 341–407. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[KLM92] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing STOC '92 (Victoria, British Columbia, Canada, May 1992),* pages 318–326, New York, 1992. ACM Press.

[Kra93] L. Krauß. Schnelle Scheduling-Algorithmen für Intervall-Graphen mit Release-Times und Deadlines. Master's thesis, Fachbereich Informatik, J.W. Goethe-Universität Frankfurt am Main, January 1993.

[KUW86] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica,* 6(1):35–48, 1986.

[KVV85] D. Kozen, U. V. Vazirani, and V. V. Vazirani. $\mathcal{NC}$ algorithms for comparability graphs, interval graphs and testing for unique perfect matching. In *Proceedings Fifth Conference on Foundations of Software Technology and Theoretical Computer Science,* LNCS 206, pages 496–503, Berlin, 1985. Springer-Verlag.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

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

[LN93] M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (San Diego, California, May 1993)*, pages 448–457, New York, 1993. ACM Press.

[LO94] R. Lin and S. Olariu. An optimal parallel matching algorithm for cographs. *J. Parallel Distrib. Comput.*, 22(1):26–36, 1994.

[LPV81] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. Comput.*, C-30(2):93–100, February 1981.

[LRK78] J. K. Lenstra and A. H. G. Rinnooy Kan. The complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.

[LVV93] J. K. Lenstra, M. Veldhorst, and B. Veltman. The complexity of scheduling trees with communication delays. In T. Lengauer, editor, *Proceedings of the First Annual European Symposium on Algorithms, ESA'93 (Bad Honnef, Germany, September 1993)*, LNCS 726, pages 284–294, Berlin, 1993. Springer-Verlag.

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

[May85] E. W. Mayr. Efficient parallel scheduling algorithms. In *19th Annual Asilomar Conference on Circuits, Systems and Computers*, November 1985.

[May88] E. W. Mayr. Parallel approximation algorithms. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988 (Tokyo, Japan, November/December 1988)*, volume 2, pages 542–551. Institute for New Generation Computer Technology, 1988.

[May96] E. W. Mayr. Scheduling interval orders in parallel. *Parallel Algorithms and Applications*, 8:21–34, 1996.

[MJ89] A. Moitra and R. C. Johnson. A parallel algorithm for maximum matching on interval graphs. In F. Ris and P. M. Kogge, editors, *Proceedings of the 1989 International Conference on Parallel Processing. Vol. 3 (Penn State University, August 1989)*, pages 114–120, University Park-London, 1989. Pennsylvania State University Press.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Ann. IEEE Symposium on Foundations of Computer Science (Portland, OR),* pages 478–489. IEEE, 1985.

[MSS96] R. H. Möhring, M. W. Schäffter, and A. S. Schulz. Scheduling jobs with communication delays: Using infeasible solutions for approximation. In J. Diaz and M. Serna, editors, *Proceedings of the 4th Annual European Symposium on Algorithms, ESA'96 (Barcelona, Spain, September 1996),* LNCS 1136, pages 76–90, Berlin, 1996. Springer-Verlag.

[MV80] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Ann. IEEE Symposium on Foundations of Computer Science (Syracuse, NY),* pages 17–27. IEEE, 1980.

[MV96] M. Morvan and L. Viennot. Parallel comparability graph recognition and modular decomposition. In C. Puech and R. Reischuk, editors, *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS 96 (Grenoble, France, February 1996),* LNCS 1046, pages 169–180, Berlin, 1996. Springer-Verlag.

[MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–120, 1987.

[Pic92] C. Picouleau. *Etude de problèmes d'optimisation dans les systèmes distribués.* PhD thesis, Université Paris VI, 1992.

[PK85] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing,* pages 14–20, Los Alamitos-Washington-Brussels-Tokyo, 1985. IEEE Computer Society Press.

[PY79] C. H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Comput.*, 8(3):405–409, August 1979.

[RS87] V. J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Appl. Math.*, 18:55–71, 1987.

[Ser91] M. Serna. Approximating linear programming is log-space complete for $\mathcal{P}$. *Inf. Process. Lett.*, 37:233–236, 1991.

[Set76] R. Sethi. Scheduling graphs on two processors. *SIAM J. Comput.*, 5(1):73–82, March 1976.

[SH93] S. Sunder and X. He. Scheduling interval ordered tasks in parallel. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science, STACS 93 (Würzburg, Germany, February 1993),* LNCS 665, pages 100–109, Berlin, 1993. Springer-Verlag.

[Sim78] B. B. Simons. A fast algorithm for single processor scheduling. In *Proceedings of the 19th Ann. IEEE Symposium on Foundations of Computer Science (Ann Arbor, MI),* pages 246–252. IEEE, 1978.

[Sim80] B. B. Simons. A fast algorithm for multiprocessor scheduling. In *Proceedings of the 21st Ann. IEEE Symposium on Foundations of Computer Science (Syracuse, NY),* pages 50–53. IEEE, 1980.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms,* 2(1):88–102, March 1981.

[SW89] B. B. Simons and M. K. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM J. Comput.*, 18:690–710, 1989.

[SX97] M. Serna and F. Xhafa. Approximating scheduling problems in parallel. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing (Passau, Germany, August 1997),* LNCS 1300, pages 440–449, Berlin, 1997. Springer-Verlag.

[TLC89] W. W. Tsang, T. W. Lam, and F. Y. L. Chin. An optimal EREW parallel algorithm for parenthesis matching. In F. Ris and P. 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.

[Tre96] L. Trevisan. Positive linear programming, parallel approximation and PCP's. In J. Diaz and M. Serna, editors, *Proceedings of the 4th Annual European Symposium on Algorithms, ESA'93 (Barcelona, Spain, September 1996),* LNCS 1136, pages 62–75, Berlin, 1996. Springer-Verlag.

[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.

[Vaz94] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. *Combinatorica,* 14:71–109, 1994.

[Ver96] J. Verriet. Scheduling interval orders with release dates and deadlines. Technical Report UU-CS-1996-12, Department of Computer Science, Utrecht University, March 1996.

[VRKL96] T. A. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler. Scheduling in and out forests in the presence of communication delays. *IEEE Transactions on Parallel and Distributed Systems,* 7(10):1065–1074, October 1996.

[VTL82] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, May 1982.

[VV85] U. Vazirani and V. V. Vazirani. The two-processor scheduling problem is in $\mathcal{RNC}$. In *Proceedings of the 17th Ann. ACM Symposium on Theory of Computing (Providence, RI),* pages 11–21, New York, 1985. ACM Press.

[War81] M. K. Warmuth. *Scheduling on Profiles of Constant Breadth.* PhD thesis, Dept. of Computer Science, University of Colorado, Boulder, 1981.

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

Index