Provably Efficient Scheduling for Languages with Fine-Grained Parallelism *

Guy E. Blelloch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
blelloch@cs.cmu.edu

Phillip B. Gibbons
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
gibbons@research.bell-labs.com

Yossi Matias
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
matias@research.bell-labs.com

Abstract

Many high-level parallel programming languages allow for fine-grained parallelism. As in the popular work-time framework for parallel algorithm design, programs written in such languages can express the full parallelism in the program without specifying the mapping of program tasks to processors. A common concern in executing such programs is to schedule tasks to processors dynamically so as to minimize not only the execution time, but also the amount of space (memory) needed. Without careful scheduling, the parallel execution on p processors can use a factor of p or larger more space than a sequential implementation of the same program.

This paper first identifies a class of parallel schedules that are provably efficient in both time and space. For any computation with w units of work and critical path length d, and for any sequential schedule that takes space s_1, we provide a parallel schedule that takes fewer than $w/p + d$ steps on p processors and requires less than $s_1 + p \cdot d$ space. This matches the lower bound that we show, and significantly improves upon the best previous bound of $s_1 \cdot p$ space for the common case where $d \ll s_1$.

The paper then describes a scheduler for implementing high-level languages with *nested* parallelism, that generates schedules in this class. During program execution, as the structure of the computation is revealed, the scheduler keeps track of the active tasks, allocates the tasks to the processors, and performs the necessary task synchronization. The scheduler is itself a parallel algorithm, and incurs at most a constant factor overhead in time and space, even when the scheduling granularity is individual units of work. The algorithm is the first efficient solution to the scheduling problem discussed here, even if space considerations are ignored.

Our results apply to a variety of memory allocation schemes in programming languages (stack allocation, explicit heap management, implicit heap management). Space allocation is modeled as various games on weighted and group-weighted directed acyclic graphs (DAGs). Our space bounds are obtained by proving properties relating parallel schedules and parallel pebble games on arbitrary DAGs to their sequential counterparts. The scheduler algorithm relies on properties we prove for planar and series-parallel DAGs.

* A preliminary version of this work appears in the *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, Calif.). ACM, New York, July 1995, pp. 1–12.

1 Introduction

Many high-level parallel programming languages encourage the use of dynamic fine-grained parallelism. Such languages include both data-parallel languages such as HPF [Hig93] and NESL [BCH+94], as well as control-parallel languages such as ID [ANP89], Sisal [FCO90] or Proteus [MNP+90]. The goal of these languages is to have the user expose the full parallelism in an algorithm, which is often much more than the number of processors that will be used, and have the language implementation schedule the fine-grained parallelism onto processors. In these languages costs can be measured abstractly in terms of the total number of operations executed by the program (the *work*) and the length of the longest sequence of dependences between operations (the critical path length or *depth*).

For example, consider the following pseudo-code for multiplying two $n \times n$ matrices a and b:

```
In parallel for i from 1 to n
    In parallel for j from 1 to n
       r[i,j] = TreeSum(In parallel for k from 1 to n
                              a[i,k]*b[k,j])
```

The program performs $\Theta(n^3)$ work and has $\Theta(\lg n)$ depth since the nested loops are all parallel and the critical path is limited by the summation, which can be organized as a computation on a tree of $\lg n$ depth.

Such fine-grained parallel languages present a high-level programming model, and often lead to much shorter and clearer code than languages which require the user to map tasks to a fixed set of processors. On the other hand, users of these languages rely heavily on the implementation to deliver good performance to their high-level codes. Understanding the performance and memory use of the languages often requires a detailed understanding of the implementation, and performance anomalies are common—heuristics used in the implementation often fail for certain programs. For instance, a natural implementation of the above pseudo-code for matrix multiplication would require $\Theta(n^3)$ space, whereas a sequential computation requires only $\Theta(n^2)$ space. (See Figure 1.) In order to obtain the same bounds for a parallel implementation, heuristic techniques that limit the amount of parallelism in the implementation have been

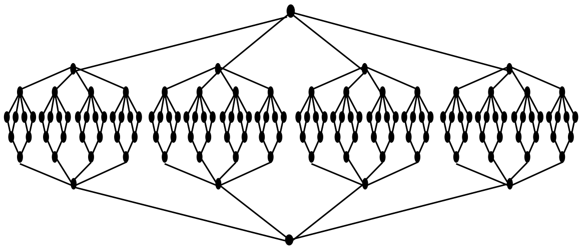


Figure 1: The task structure of a computation for multiplying two $n \times n$ matrices (for $n = 4$), represented as a directed acyclic graph. Nodes represent unit-work tasks, and edges (assumed to be directed downward in the figure) represent control and/or data flow between the tasks. A level-by-level schedule of this graph requires $\Theta(n^3)$ space for program variables, in order to hold the n^3 intermediate results required at the widest level of the graph. Moreover, such a schedule may use $\Theta(n^3)$ space for task bookkeeping, in order to keep track of tasks ready to be scheduled. Note that the standard depth-first sequential schedule of this graph uses only $\Theta(n^2)$ space, counting the space for the input and output matrices.

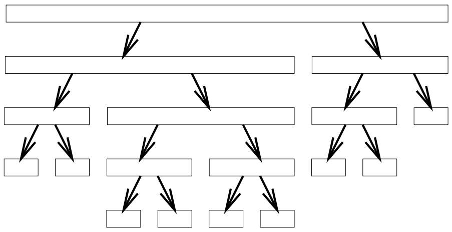


Figure 2: The high-level DAG structure of a quicksort computation. Each block represents a sub-DAG that splits the data into its lesser and greater elements. The sub-DAGs on n inputs can be implemented in size $O(n)$ and depth $O(\lg n)$, using prefix sums, for example. This leads to a DAG with expected work (size) $O(n \lg n)$ and expected depth $O(\lg^2 n)$. Since the split sizes are not know ahead of time, the structure of the DAG unfolds dynamically, which requires an online scheduler.

used [BS81, Hal85, RS87, CA88, JP92], but these are not guaranteed to be space efficient in general.

There have been several recent works [BL93, BL94, BS94, Bur96] presenting scheduling algorithms with guaranteed performance bounds, both in terms of time and space. Blumofe and Leiserson [BL93, BL94] consider the class of *fully-strict* computations, and show that a computation with w work and d depth that requires s_1 space when executed using a (standard) depth-first sequential schedule can be implemented in $O(w/p + d)$ time and $s_1 \cdot p$ space on p processors. Similar space bounds were obtained in [BS94, Bur96], for a different class of parallel programs.

This paper presents new scheduling techniques that significantly improve upon these previous results. As with much previous work we model computations as directed acyclic graphs (DAGs). The nodes in the DAG represent unit-time tasks or *actions*, and the edges represent any ordering dependences between the actions that must be respected by the implementation. As in [BL93, BL94, BS94, Bur96], we focus primarily on programs for which the DAG is independent of the order in which tasks are scheduled/executed.[1] The work of a computation corresponds to the number of nodes in the DAG, and the depth corresponds to the longest path in the DAG.

A scheduler is responsible for mapping each action to a (time step, processor) pair such that each processor has at most one task per time step and no dependence is violated. An *offline* scheduler has knowledge of the entire DAG prior to the start of the parallel program. An *online* scheduler, in contrast, learns about the structure of the DAG only as the computation proceeds, and must make scheduling decisions online based on only partial knowledge of the DAG. Matrix multiplication is an example of a computation amenable to offline scheduling, since its structure depends only on the size of the input. Quicksort, shown in Figure 2, is an example of a computation requiring online scheduling, as the shape of its DAG is revealed only as the computation proceeds.

We present results for both offline and online scheduling that improve upon previous results [BL93, BL94, BS94] in several ways. First we prove space bounds of $s_1 + O(p \cdot d)$ memory words while maintaining an $O(w/p+d)$ time bound. This matches a lower bound that we show, and significantly improves upon the previous results of $s_1 \cdot p$ space, when $d \ll s_1$. Note that for most parallel computations, $d \ll s_1$, since typically the critical path length, d, is much smaller than the size of the input, n, and, since s_1 includes the space for the input, $n \leq s_1$. Second our results apply to a more flexible model for memory allocations. We allow for arbitrary heap allocation/deallocation, both in the case that it is explicitly handled by the programmer and when it is implicitly handled by a garbage collector. (The model in [BL93, BL94] allows for only stack allocation; the model in [BS94] allows for only explicit heap allocation/deallocation.) Third, unlike [BL93, BL94], we allow for arbitrary out-degree when spawning new tasks, rather than restricting to constant out-degree.

On the other hand, the previous results have several important advantages over our results. The scheduling algorithms in [BL94, BS94] use distributed "work stealing", in contrast to our centralized, parallel approach. This lowers the scheduling overheads, since the overheads are incurred only by the processors attempting to find tasks and the processors contacted in order to find tasks. As long as each processor has tasks to execute, no overheads are incurred. Moreover, Blumofe and Leiserson [BL94] consider not just time and space bounds but also communication bounds. Their scheduler is shown to provide good bounds on the total amount of interprocessor communication for the task model they consider.

For offline scheduling, our results apply to any DAG and are relative to any sequential schedule (*i.e.*, given any sequential schedule on a DAG with w work and d depth, we can construct a p-processor schedule of fewer than $w/p + d$ steps whose space bound is within an additive $O(p \cdot d)$ term of the sequential space). For online scheduling, our results can be applied as long as the scheduler can keep tasks that are ready to execute in an appropriate priority order. We show how

[1] Programs for which the dag does not depend on the scheduling order are called *deterministic* programs in the scheduling literature; the remaining programs are called *nondeterministic*. Deterministic programs include programs that make use of randomization, as long as the randomization does not affect the dag.

this order can be maintained for *nested-parallel* languages, and describe an implementation of an online scheduler.[2] The implementation maintains the space and time bounds within a constant factor including all costs for scheduling and synchronization, even when the scheduling granularity is individual unit-time actions.

The following outlines the paper and its main results. Each section relies on the previous sections.

Premature Nodes (Section 2): We consider a class of parallel list schedules that do not diverge significantly from the sequential schedules on which they are based. A p-schedule of a DAG is a scheduling of tasks that executes at most p actions on each time step. We define a p-schedule based on a 1-schedule to be a list schedule such that the priority among the tasks for the p-schedule is given by their order in the 1-schedule. Next, we introduce the notion of premature nodes, where a premature node in a parallel schedule is one that is scheduled prematurely (out-of-order) relative to a sequential schedule. The main result of the section is the following bound:

- For any DAG of depth d, and for any p-schedule $\mathcal{S}$ based on any sequential schedule $\mathcal{S}_1$, the maximum number of premature nodes in $\mathcal{S}$ with respect to $\mathcal{S}_1$ is at most $(p-1)\cdot(d-1)$.

We also prove a lower bound which matches the upper bound.

In the remaining sections we use the upper bound on premature nodes to prove our space bounds. In brief, when comparing the space, s_p, used by a parallel schedule, $\mathcal{S}$, with the space, s_1, used by a sequential schedule, $\mathcal{S}_1$, the premature nodes at any step in $\mathcal{S}$ represent (the only) opportunities for $\mathcal{S}$ to allocate space prematurely relative to $\mathcal{S}_1$. Thus by bounding the number of premature nodes, and then bounding (to a constant) the space that each such node can allocate, we will show that $s_p \leq s_1 + O(pd)$.

Space Models (Section 3): We consider a variety of space models in an attempt to capture the variation of space allocation schemes used in languages with fine-grained parallelism. The first space model is the parallel pebble game of Savage and Vitter [SV84], a variant of the standard pebble game [HPV77, Pip80] in which pebbles can be placed on p nodes of the computation DAG at each step instead of just 1. Each pebble corresponds to a unit of allocated space needed to hold an intermediate result of the computation, and pebbles can be removed (and reused) as soon as the intermediate result is no longer needed. The goal is to minimize the total number of pebbles used to traverse the DAG within a certain number of steps.

In addition, we consider space models based on weighting DAG nodes with the amount of memory they allocate (positive weight) or deallocate (negative weight) and keeping track of the accumulated weight as the schedule proceeds. We study of family of such models, in increasing order of generality.

We describe how these space models capture various allocations schemes supported by programming languages, including the use of stack frames, explicit heap management with allocate and free, and implicit heap management with garbage collection. The main results of the section are:

- If a DAG G with n nodes and depth d can be 1-pebbled with s_1 pebbles in n steps, then it can be p-pebbled with fewer than $s_1 + p\cdot d$ pebbles in $n/p + d$ steps.
- Consider a weighted DAG G with n nodes and depth d. Given any 1-schedule $\mathcal{S}_1$ of G that requires s_1 space (maximum accumulated weight), the greedy p-schedule based on $\mathcal{S}_1$ will require less than $s_1 + p\cdot d$ space and will take at most $n/p + d$ steps.

The first result is the first non-trivial general result relating sequential and parallel pebble games. The second result assumes positive weights are at most one. For arbitrary positive weights, the space bound can be obtained by incurring extra work (extra steps). We also prove lower bounds that match within a constant factor. The results in this section apply to any DAG and are relative to any sequential schedule or pebbling. They can be used for offline scheduling, with provably good time and space bounds.

Online Schedules (Section 4): To implement programming languages it is necessary to consider online schedules in which the DAG associated with a computation is not known ahead of time. Furthermore we are interested in basing the p-schedule on the particular schedule executed by a "standard" serial implementation. This is most typically a depth-first schedule (1DF-schedule).[3] We call the p-schedule based on a 1DF-schedule a PDF-schedule. To execute a PDF-schedule online we need only maintain the actions that are ready to be executed, prioritized according to their 1DF-schedule. We define a simple online scheduling algorithm called the P-stack algorithm, and show the following:

- The online P-stack scheduling algorithm implements a greedy PDF-schedule for planar DAGs.

This result implies that the space bounds from the previous section apply to this online problem.

Planar DAGs account for a large class of parallel languages including all nested-parallel languages, as well as other languages such as Cilk [BJK+95]. In the next section we consider a particular type of planar DAG called a series-parallel DAG, which is sufficient for modeling all nested-parallel languages.

Nested Parallel Implementation (Section 5): This section describes the full details of implementing a PDF-schedule for a class of parallel languages, and proves both time and space bounds for the implementation. Our time and space bounds include all the costs for the scheduler as well as the costs for the computation itself.

The two main tasks of the scheduler are to identify nodes (*i.e.*, actions) that are ready to be executed and to maintain the P-stack data structure. We consider three implementations of nested-parallel languages that allow for fork-join constructs and/or parallel loops. The first implementation assumes that the degree of each node is constant. This can model languages in which each task can fork at most a constant number of new tasks at a time. The second places no limit on the degree of nodes, and hence can

[2] The restriction to nested-parallel languages is similar to the restriction to fully-strict computations made by Blumofe and Leiserson. The difference is discussed in Section 6.

[3] An exception to this is in the implementation of lazy languages in which computations are executed on demand.

model parallel loops; this implementation employs a fetch-and-increment operation. The main concern is in bounding the size of the P-stack data structure despite the arbitrary fanout. The final implementation avoids the use of the fetch-and-increment by carefully managing the tasks and using a scan operation (parallel prefix) instead. The main concern in this implementation is efficiently synchronizing the tasks. The strongest result is based on the final implementation:

- Consider any computation expressed in a nested-parallel language which does w work, has depth d, uses sequential space s_1, and allocates at most $O(w)$ space. This computation can be implemented on a PRAM with prefix-sums (*i.e.*, on the scan model [Ble89]) in $O(w/p+d)$ time and $s_1+O(p\cdot d)$ space, accounting for all computation, scheduling and synchronization costs.

Since a prefix-sum can be implemented work-efficiently in $O(\lg p)$ time on any of the PRAM models, this implies bounds of $O(w/p + d\cdot \lg p)$ time and $s_1 + O(p\cdot d\cdot \lg p)$ space on a PRAM without prefix-sums. The scheduler itself only requires exclusive-read exclusive-write (EREW) capabilities, although the computation might require a more powerful PRAM depending on the type of concurrent access allowed by the language. The restriction that a computation performing w work allocates at most $O(w)$ space is a natural one, since the computation can only read or write to w memory words in w work. However, computations that use memory as a bulletin board could allocate more than $O(w)$ space.

Related Work and Discussion (Sections 6 and 7): These sections present a more detailed comparison with related work and a discussion of several important issues including extensions to handle nondeterministic programs and practical implementations of our scheduling algorithm.

2 Parallel schedules with a premature nodes bound

In this section, we consider the well-studied parallel DAG scheduling problem, and prove a general property relating a class of parallel schedules to their associated sequential schedules. We first define a class of parallel schedules that are "based on" given sequential schedules, such that the sequential schedule dictates the scheduling priorities to the parallel schedule. The parallel schedule, although based on a given sequential schedule, will almost always schedule nodes out-of-order (*i.e.*, prematurely) with respect to the sequential schedule, in order to achieve the desired parallelism at each step. Then, in our main theorem, we show that the number of these "premature" nodes at any step of the p-processor schedule is bounded by p times the depth, d, of the DAG. This theorem is the key ingredient in our space bound proofs of the next section.

Graph and DAG scheduling terminology. We begin by defining the terminology used in this section and throughout the paper. We use standard graph terminology (see, *e.g.*, [CLR90]), which, for completeness, is reviewed below. A directed graph $G=(V,E)$ consists of a set of vertices V and a set of edges $E\subseteq V\times V$, where each edge $(u,v)\in E$ is *outgoing* from node u and *incoming* to node v. A (directed, simple) *path* from v_1 to v_k, $k\geq 1$, in a directed graph is a sequence of distinct nodes $v_1,\ldots,v_k$ such that there is a directed edge (v_i,v_{i+1}) for all $i<k$. A directed graph has a (directed) *cycle* if there exists nodes u and v such that there is a path from u to v and an edge (v,u). A *directed acyclic graph (DAG)* is a directed graph with no cycles. The *roots* of a DAG are the nodes with no incoming edges; the *leaves* of a DAG are the nodes with no outgoing edges. The *depth* or *level* of a node v in a DAG is the number of nodes on the longest path from a root node to v, inclusive. The depth of a DAG is the maximum depth of any node in the DAG.

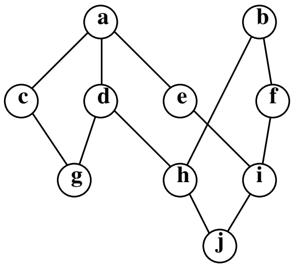
Let u and v be nodes in a DAG G. Node u is a *parent* of node v, and v is a *child* of u, if there is an edge (u,v) in G. Node u is an *ancestor* of node v, and v is a *descendant* of u, if there is a path in G from u to v. Nodes u and v are *ordered* if u is an ancestor or a descendant of v; otherwise they are *unordered*. For a DAG $G=(V,E)$, an edge $(u,v)\in E$ is *transitive* (or redundant) if there is another path in G from u to v. The *transitive reduction* of G is the DAG induced by removing all transitive edges.

In the models discussed in this paper each node of the DAG represents a unit-time *action*. The edges represent any ordering dependences between the actions—a path from a node u to a node v implies that the action for u must complete before the action for v starts. Tasks of greater than unit-time duration can be modeled as a sequence of actions on a path. The DAG may depend on the input, but it does not depend on the order in which the nodes are scheduled. (This models deterministic programs; extensions of the models to handle nondeterministic programs are discussed briefly in Section 7.) Scheduling on such DAGs has been called unit execution time (UET) scheduling [Cof76].

A *schedule* (or UET schedule) of a DAG G is a sequence of $\tau\geq 1$ steps, where each step i, $i=1,\ldots,\tau$, defines a set of nodes V_i (that are *visited*, or scheduled at this step) such that the following two properties hold. First, each node appears exactly once in the schedule, *i.e.*, the sets V_i, $i=1,\ldots,\tau$, partition the set of nodes of G. Second, a node is scheduled only after all its ancestors have been scheduled in previous steps, *i.e.*, if $v\in V_i$ and u is an ancestor of v, then $u\in V_j$ for some $j<i$. Thus if multiple nodes are scheduled in a step, these nodes are unordered. A *p-schedule*, for $p\geq 1$, is a schedule such that each step consists of at most p nodes, *i.e.*, for $i=1,\ldots,\tau$, $|V_i|\leq p$. A schedule is *parallel* if $p>1$, otherwise it is *sequential*. Let $|\mathcal{S}|$ be the number of steps in a schedule $\mathcal{S}$. An example DAG and schedule are shown in Figure 3.

Consider a schedule $\mathcal{S}=V_1,\ldots,V_\tau$ of a DAG G. For $i=0,\ldots,\tau$, let $C_i=V_1\cup\cdots\cup V_i$ be the *completed set* after step i, *i.e.* the set of nodes in the first i steps of the schedule. Figure 3 depicts the completed sets for an example schedule. A node $v\in G$ is *scheduled* prior to a step i in $\mathcal{S}$, $i=1,\ldots,\tau$, if v appears in C_{i-1}. A schedule induces a partial order on the nodes in each of its completed sets C_i: node $u\in V_j$ *precedes* node $v\in V_k$ if and only if $j<k\leq i$. Thus a 1-schedule induces a total order on the nodes in each of its completed sets. It is natural to view the steps of a schedule as sequenced in time; thus if u precedes v, then u is *earlier* than v and v is *later* or *more recently scheduled* than u. An unscheduled node v is *ready* at step i in $\mathcal{S}$ if all its ancestors (equivalently, all its parents) are scheduled prior to step i.

In a *breadth-first* or *level-order* 1-schedule, a node is scheduled only after all nodes at lower levels are scheduled. A *depth-first* 1-schedule (1DF-schedule) is defined as follows. At each step, if there are no scheduled nodes with a ready child, schedule a root node, otherwise schedule a ready child of the most recently scheduled node with a ready child. The



$$
\begin{array}{ll}
V_1 = \{a\} & C_1 = \{a\} \\
V_2 = \{b, c, d, e\} & C_2 = \{a, b, c, d, e\} \\
V_3 = \{f, h\} & C_3 = \{a, b, c, d, e, f, h\} \\
V_4 = \{g, i\} & C_4 = \{a, b, c, d, e, f, g, h, i\} \\
V_5 = \{j\} & C_5 = \{a, b, c, d, e, f, g, h, i, j\}
\end{array}
$$

Figure 3: An example DAG and schedule on that DAG. The DAG edges are assumed to be directed downwards. For $i = 1, \ldots, 5$, V_i is the set of nodes scheduled at step i and C_i is the completed set after step i.

node scheduled at step i in a 1DF-schedule is said to have 1DF-*number* i.

We will be particularly interested in (single source and sink) *series-parallel* DAGs, which are defined inductively, as follows: The graph, G_0, consisting of a single node (which is both its source and sink node) and no edges is a series-parallel DAG. If G_1 and G_2 are series-parallel, then the graph obtained by adding to $G_1 \cup G_2$ a directed edge from the sink node of G_1 to the source node of G_2 is series-parallel. If $G_1, \ldots, G_k$, $k \geq 1$, are series-parallel, then the graph obtained by adding to $G_1 \cup \cdots \cup G_k$ a new source node, u, with a directed edge from u into the source nodes of $G_1, \ldots, G_k$, and a new sink node, v, with a directed edge from the sink nodes of $G_1, \ldots, G_k$ into v is series-parallel. Thus, a node may have indegree or outdegree greater than 1, but not both. We say that the source node, u, is the *lowest common source node* for any pair of nodes $w \in G_i$ and $w' \in G_j$ such that $i \neq j$.

Greedy parallel schedules. The following well-known fact places a lower bound on the number of steps in any p-schedule:

Fact 2.1 *For all $p \geq 1$, any p-schedule of a* DAG *G with n nodes and depth d requires at least* $\max\{n/p, d\}$ *steps.*

This lower bound is matched within a factor of two by any "greedy" p-schedule. A *greedy p-schedule* is a p-schedule such that at each step i, if at least p nodes are ready, then $|V_i| = p$, and if fewer than p are ready, then V_i consists of all the ready nodes. Generalizing previous results [Gra66, Gra69, Bre74], Blumofe and Leiserson [BL93] showed the following:

Fact 2.2 *For any* DAG *of $n \geq 1$ nodes and depth d, and for any $p \geq 1$, the number of parallel steps in any greedy p-schedule is less than $n/p + d$.*

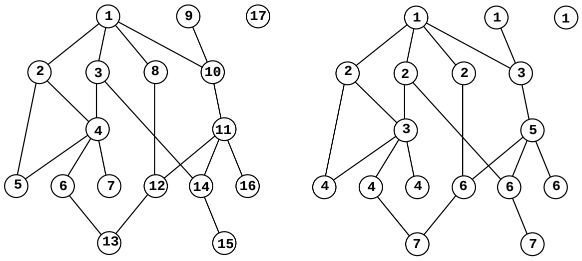


Figure 4: A greedy PDF-schedule of a DAG G, for $p = 3$. On the left, the nodes of G are numbered in order of a 1DF-schedule, $\mathcal{S}_1$, of G. On the right, G is labeled according to the greedy PDF-schedule, $\mathcal{S}$, based on $\mathcal{S}_1$; $\mathcal{S} = V_1, \ldots, V_7$, where for $i = 1, \ldots, 7$, V_i, the set of nodes scheduled in step i, is the set of nodes labeled i in the figure.

2.1 An important class of parallel schedules

This paper provides parallel schedules for DAGs with provably good time and space bounds relative to sequential schedules of the same DAG, improving the bounds obtained by previous works. A key ingredient to our improved results is the identification of the following class of parallel schedules.

Definition (p-schedule based on a 1-schedule) *We define a p-schedule, $\mathcal{S}$, to be* based on *a 1-schedule, $\mathcal{S}_1$, if, at each step i of $\mathcal{S}$, the k_i earliest nodes in $\mathcal{S}_1$ that are ready at step i are scheduled, for some $k_i \leq p$.*

In other words, for all ready nodes u and v, if u precedes v in $\mathcal{S}_1$, then either both are scheduled, neither are scheduled, or only u is scheduled.

A p-schedule based on a 1-schedule is a particular kind of list schedule. In a list schedule, all the tasks are listed in a fixed priority order prior to scheduling, and at each step, the ready tasks with the highest priorities are scheduled [Gra66, Cof76]. In our case, the list is, in fact, a 1-schedule. Later, in Section 4, we consider "online" scenarios in which the list is revealed only as the computation proceeds, and lower priority tasks must often be scheduled before certain higher priority tasks are even revealed.

The motivation for considering a p-schedule based on a 1-schedule is that it attempts to deviate the least from the 1-schedule, by seeking to make progress on the 1-schedule whenever possible. Intuitively, the closer the p-schedule is to the 1-schedule, the closer its resource bounds may be to the 1-schedule.

Note that given a 1-schedule, the greedy p-schedule based on the 1-schedule is uniquely defined, and will have less than $n/p + d$ steps for a DAG of n nodes and depth d.

A p-schedule that we consider in Section 4 is the depth-first p-schedule. A *depth-first p-schedule* (PDF-schedule) is a p-schedule based on a depth-first 1-schedule. An example is given in Figure 4.

Consider a DAG G with n nodes and depth d. In general, any p-schedule that schedules more than one node in most of its step (*i.e.*, uses far fewer than n steps) will schedule nodes out-of-order ("prematurely") with respect to almost all 1-schedules of G. An important property of p-schedules based on 1-schedules is that, for such schedules, we can prove that the number of nodes that are simultaneously premature is bounded by $(p-1) \cdot (d-1)$.

2.2 Tight bounds on premature nodes

Let $\mathcal{S}_1$ be a 1-schedule and $\mathcal{S}$ be a p-schedule for the same DAG. For each completed set, C, of $\mathcal{S}$, let C_i^1 be the longest prefix of $\mathcal{S}_1$ contained within C, defined as follows.

Definition (largest contained 1-prefix) *C_i^1 is the* largest contained 1-prefix *of C if C_i^1 is a completed set of $\mathcal{S}_1$, $C_i^1 \subseteq C$, and either i is the last step of $\mathcal{S}_1$ or the node scheduled at step $i+1$ of $\mathcal{S}_1$ is not in C (thus $C_{i+1}^1 \not\subseteq C$).*

Next, we define the "premature nodes" for a completed set C of $\mathcal{S}$. If we list the nodes in order of $\mathcal{S}_1$ and then mark the ones occuring in C, then the premature nodes will be all the marked nodes following the first unmarked node. Formally, we have:

Definition (premature nodes) *Let C be a completed set and C_i^1 be the largest contained 1-prefix of C. The set of* premature nodes *in C, $\mathcal{P}(C)$, is $C - C_i^1$.*

Figure 5 shows the premature nodes in the first six completed sets of the PDF-schedule of Figure 4.

Definition (maximum number of premature nodes) *The* maximum number of premature nodes *in $\mathcal{S}$ with respect to $\mathcal{S}_1$ is* $\max\{|\mathcal{P}(C)| : C$ *is a completed set of* $\mathcal{S}\}$.

The following theorem gives an upper bound on the maximum number of premature nodes. This is the key theorem for the results in this paper, and may be of independent interest.

Theorem 2.3 (upper bound on premature nodes) *For any DAG of depth $d \geq 1$, and for any 1-schedule $\mathcal{S}_1$, the maximum number of premature nodes with respect to $\mathcal{S}_1$ in any p-schedule based on $\mathcal{S}_1$ is at most $(p-1)\cdot(d-1)$.*

Proof. The main ideas of the proof are as follows.

- Premature nodes are scheduled at a step only when all other (nonpremature) nodes are not ready (since nonpremature nodes have priority).
- All unscheduled nonpremature nodes at the smallest level containing such nodes are ready (since their parents are nonpremature and have been scheduled).
- Thus any step that schedules a premature node also completes the smallest level. This can happen at most $d-1$ times, with at most $p-1$ premature nodes being scheduled each time.

Accordingly, let G be a DAG of depth d, and let $\mathcal{S}$ be a p-schedule based on the given 1-schedule $\mathcal{S}_1$. We will show that each completed set of $\mathcal{S}$ has at most $(p-1)\cdot(d-1)$ premature nodes with respect to $\mathcal{S}_1$. Specifically, consider an arbitrary completed set, C, of $\mathcal{S}$. Let C^1 be the completed set of $\mathcal{S}_1$ that is the largest contained 1-prefix of C. We will show that

$$|\mathcal{P}(C)| = |C| - |C^1| \leq (p-1)\cdot(d-1) \ .$$

Let v be the first node in $\mathcal{S}_1$ that is not in C^1. (If no such v exists, then C^1 is all the nodes of G, and $|\mathcal{P}(C)| = 0$.) By definition of C^1, node v is the highest priority node not in C. Since $\mathcal{S}_1$ is a schedule, all the parents of v are in C^1, at levels less than d.

The nodes in C^1 can be partitioned by level: for $\ell = 1,\ldots,d$, let $C^1[\ell]$ be the set of nodes in C^1 at level ℓ. We say a step j in $\mathcal{S}$ *completes* a level ℓ' if

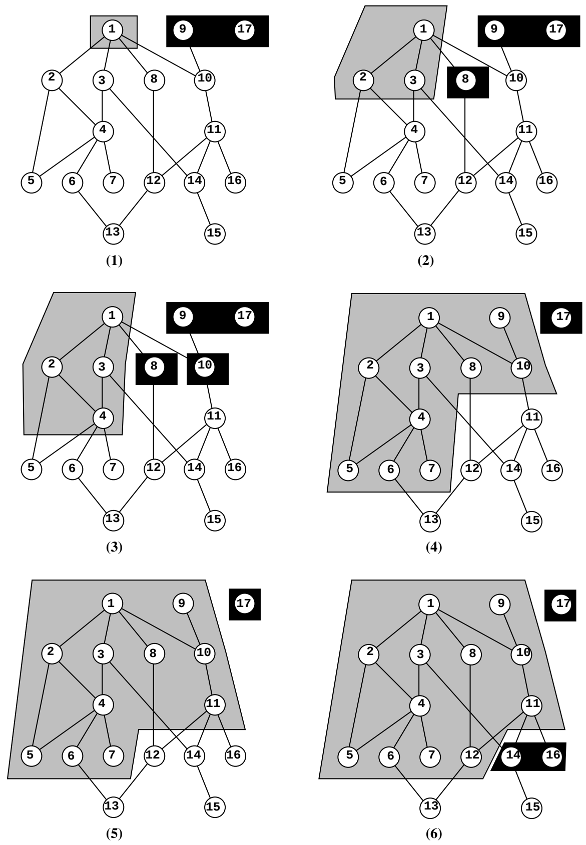


Figure 5: Premature nodes for the DAG in Figure 4, its 1DF-schedule $\mathcal{S}_1$, and the PDF-schedule $\mathcal{S} = V_1,\ldots,V_7$ based on $\mathcal{S}_1$. Shown are the premature nodes with respect to $\mathcal{S}_1$ in the completed sets $C_1,\ldots,C_6$, respectively. For each completed set, the nodes are numbered according to $\mathcal{S}_1$, the nodes in the largest contained 1-prefix of $\mathcal{S}_1$ are marked in grey, and the premature nodes are marked in black. The maximum number of premature nodes for this schedule is 4, as required by completed set C_3.

1. $C^1[\ell'] \not\subseteq C_{j-1}$, but
2. $C^1[\ell] \subseteq C_j$ for $\ell = 1,\ldots,\ell'$.

For example, for the schedule depicted in Figure 5, consider the completed set C after 3 steps of $\mathcal{S}$: $C = \{1,2,3,4,8,9,10,17\}$, the set of nodes marked in either grey or black in diagram 3 of the figure. For this completed set, $C^1 = \{1,2,3,4\}$, the set of nodes marked in grey. We have that $C^1[1] = \{1\}$, $C^1[2] = \{2,3\}$, $C^1[3] = \{4\}$, and $v = 5$, the first node in $\mathcal{S}_1$ not in C^1. The proof will show that the number of nodes marked in black is at most $(p-1)\cdot(d-1)$, which for this example equals 8, and indeed there are only four such nodes in the figure.

We first show that any step j of $\mathcal{S}$ which contains at least one node in $\mathcal{P}(C)$ completes a level $\ell' < d$ of C^1. Suppose there is no level $\ell < d$ such that $C^1[\ell] \not\subseteq C_{j-1}$. Then all nodes at levels less than d are in C_{j-1}, including all the parents of v. But then at step j, v would be a higher priority ready node than any node in $\mathcal{P}(C)$, and hence would be in C_j, a contradiction. Therefore, consider the smallest level $\ell' < d$ such that $C^1[\ell'] \not\subseteq C_{j-1}$, and let $U = C^1[\ell'] - C_{j-1}$, the nonempty set of nodes in $C^1[\ell']$ that are not in C_{j-1}. In the example, for $j = 1$, we have $\ell' = 1$ and $U = \{1\}$, for $j = 2$, $\ell' = 2$ and $U = \{2, 3\}$, and for $j = 3$, $\ell' = 3$ and $U = \{4\}$.

All nodes in C^1 at levels smaller than ℓ' were scheduled prior to step j, but the nodes in U were not scheduled prior to step j. Since C^1 is the completed set of a schedule, all parents of nodes in $C^1[\ell']$ (if any) are also in C^1, at levels smaller than ℓ'. Thus all nodes in U were ready at step j. Any node in C^1 has higher priority than any node not in C^1, thus nodes not in C^1 would be scheduled at step j only if all ready nodes in C^1 were also scheduled at step j. Thus, since nodes not in C^1 were scheduled at step j, then all nodes in U were also scheduled at step j, and hence step j completes level ℓ' of C^1. In the example, step $j = 1$ completes $C^1[1]$, step $j = 2$ completes $C^1[2]$, and step $j = 3$ completes $C^1[3]$.

Since G has d levels, there can be at most $d - 1$ steps of $\mathcal{S}$ that complete a level less than d. Each such step has at least one node in C^1, so it may have at most $p - 1$ nodes in $\mathcal{P}(C)$. It follows that $|\mathcal{P}(C)| \leq (p - 1) \cdot (d - 1)$. ∎

Note that the upper bound applies to any p-schedule based on a 1-schedule, greedy or otherwise.

The following theorem shows that this upper bound is tight for *any* greedy p-schedule. Specifically, it shows that for any p and d, there exists a DAG of depth d and a 1-schedule such that the maximum number of premature nodes in any greedy p-schedule matches the upper bound.

Theorem 2.4 (lower bound on premature nodes) *For all $p \geq 1$ and $d \geq 1$, there exists a* DAG, *G, of depth d, such that for any greedy p-schedule of G, the maximum number of premature nodes with respect to any* 1DF*-schedule of G is $(p - 1) \cdot (d - 1)$.*

Proof. Let G be a forest comprised of p disjoint paths of depth d. That is, G has p root nodes, $v_{1,1}, v_{1,2}, \ldots, v_{1,p}$, and at each level $\ell = 2, \ldots, d$, there are p nodes, $v_{\ell,1}, v_{\ell,2}, \ldots, v_{\ell,p}$, such that for $k = 1, \ldots, p$, $v_{\ell,k}$ is the only child of $v_{\ell-1,k}$. Any 1DF-schedule, $\mathcal{S}_1$, of G begins by scheduling $v_{1,k'}, v_{2,k'}$, $\ldots, v_{d,k'}$ for some k'. Any greedy p-schedule, $\mathcal{S}$, of G must schedule p nodes at each step with at least p ready nodes. Thus, it must schedule all p root nodes at the first step, all p (newly ready) nodes at level 2 at the second step, and so on. Consider the completed set C_{d-1} of $\mathcal{S}$; this completed set consists of all nodes in the first $d - 1$ levels of G. Since $v_{d,j'}$ is not in C_{d-1}, $C^1_{d-1} = \{v_{1,k'}, \ldots, v_{d-1,k'}\}$ is the largest contained 1-prefix of C_{d-1}. Thus

$$|\mathcal{P}(C_{d-1})| = |C_{d-1}| - |C^1_{d-1}| = (p - 1) \cdot (d - 1) \ ,$$

which is the maximum number of premature nodes in $\mathcal{S}$. ∎

Even when the class of DAGs is restricted to series-parallel DAGs, the lower bound is $\Omega(p \cdot d)$ for any greedy p-schedule:

Theorem 2.5 *For all $p \geq 1$ and $d \geq 3$, there exists a series-parallel* DAG, *G, of depth d, such that for any greedy p-schedule of G, the maximum number of premature nodes with respect to any* 1DF*-schedule of G is $(p - 1) \cdot (d - 3)$.*

Proof. The argument is nearly identical to the proof of Theorem 2.4, except that we add a common root node and a common leaf node to the DAG to make it series-parallel. Specifically, G has a single root node, v_1, with p children, $v_{2,1}, v_{2,2}, \ldots, v_{2,p}$, and at each level $\ell = 3, \ldots, d - 1$, there are p nodes, $v_{\ell,1}, v_{\ell,2}, \ldots, v_{\ell,p}$, such that for $k = 1, \ldots, p$, $v_{\ell,k}$ is the only child of $v_{\ell-1,k}$. Finally there is a single leaf node, v_d, that is the child of $v_{d-1,1}, \ldots, v_{d-1,p}$. Clearly, G is series-parallel.

Any 1DF-schedule, $\mathcal{S}_1$, of G begins by scheduling $v_1, v_{2,k'}$, $v_{3,k'}, \ldots, v_{d-1,k'}$ for some k'. Any greedy p-schedule, $\mathcal{S}$, of G must schedule v_1 at the first step, all p (newly ready) nodes of level 2 at the second step, and so on. Consider the completed set C_{d-2} of $\mathcal{S}$. $C^1_{d-2} = \{v_1, v_{2,k'}, \ldots, v_{d-2,k'}\}$ is the largest contained 1-prefix of C_{d-2}. Thus

$$|\mathcal{P}(C_{d-2})| = |C_{d-2}| - |C^1_{d-2}| = (p - 1) \cdot (d - 3) \ ,$$

which is the maximum number of premature nodes in $\mathcal{S}$. ∎

3 Space models and bounds

We are interested in modeling the amount of space used by parallel computations, where we assume that space can be taken from and returned to a shared pool. We consider two types of space models that can be used in conjunction with DAGs: a parallel variant of a standard pebble game and a class of weighted DAG models. We prove bounds on parallel space for each of these models. The advantage of the weighted DAG models is that they allow one to consider only the transitive reduction of a DAG, while the pebble game gives more information about the relationship of where in a program a particular unit of memory is allocated to where in the program it can be deallocated. We also show how the pebble game can be converted into a weighted DAG model.

Ultimately we are interested in using the space models to account for the use of space in programming languages. The goal is to account for all space including both space for data and for control, and to account for a variety of memory management schemes, including stack allocation, explicit heap management (*e.g.* `malloc` and `free` in C), and implicit heap management with garbage collection (as in Lisp). In Section 3.3 we describe how the space models can be used to model these management schemes. Later, in Section 5, we will give details on how space for control structures (*i.e.*, the space for the scheduler itself) can be included within our space bounds.

3.1 Relating parallel pebble games to sequential ones

We consider a class of space models which is known in the literature as *pebble games*. Pebble games are common tools in studying the space requirements and the time-space trade-offs in computations (see [Pip80] for a survey). A standard (one player) pebble game [Pip80] is played on a DAG which represents a computation, as discussed in Section 2. At any point in the pebble game, some nodes of the DAG will have pebbles on them (one pebble per node), while the remaining nodes will not.

At each step of the pebble game, a pebble can be placed on any empty node which is a root of the DAG or such that all its parents are pebbled, and any number of pebbles can be removed from pebbled nodes. A pebble game is *complete* if each of the DAG nodes has been pebbled at least once during the game (equivalently, after each of the leaves of the DAG has been pebbled at least once). In such a game the pebbles represent space usage and the steps represent time. For a given DAG the two-fold objective is to find a complete pebble game which (i) has the smallest number of steps; and (ii) uses the minimal number of pebbles.

To study space requirements in parallel computation, we consider the parallel version of the pebble game, due to Savage and Vitter [SV84], which we call the *p-pebble game*. At each step of the p-pebble game, p pebbles or less can be placed on any subset of the DAG nodes (one pebble per node) which are roots of the DAG or such that all their parents are pebbled, and any number of pebbles can be removed from pebbled nodes. As with the pebble game, a p-pebble game becomes *complete* after each of the DAG nodes has been pebbled at least once (equivalently, after each of the leaves of the DAG has been pebbled at least once). Thus, a 1-pebble game is the same as the pebble game defined above.

In this paper we are interested in pebble games based on schedules—*i.e.*, for a schedule $\mathcal{S} = V_1, \ldots, V_\tau$, the i^{th} step of the game will pebble the nodes in V_i. We will use the notation $Pebbles(G, \mathcal{S})$ to denote the minimum number of pebbles that are required by a schedule $\mathcal{S}$ (either sequential or parallel) on the DAG G. To determine $Pebbles(G, \mathcal{S})$, we can simply remove a pebble when all its children have been pebbled since each node will be pebbled exactly once. We now relate p-pebble games to pebble games based on sequential schedules.

Theorem 3.1 (upper bounds for pebble games) *Let G be a DAG of n nodes and depth d, and suppose that there is a pebble game for G which uses n steps and s_1 pebbles. Then for all $p \geq 1$, there is a p-pebble game for G which takes at most $n/p + d$ steps, and which uses fewer than $s_1 + p \cdot d$ pebbles.*

Proof. Any pebble game for G that uses n steps must be based on a sequential schedule, $\mathcal{S}_1$, since each node will need to be pebbled exacly once. Let $\mathcal{G}_1$ be the optimal sequential pebble game for G that corresponds to $\mathcal{S}_1$ (the one yielding $Pebbles(G, \mathcal{S}_1)$ pebbles and based on removing a pebble as soon as all children have been pebbled). Let $\mathcal{S}$ be a greedy p-schedule based on $\mathcal{S}_1$, and let $\mathcal{G}_p$ be the optimal p-pebble game for G that corresponds to $\mathcal{S}$. The bound on the number of steps follows from Fact 2.2.

Consider an arbitrary step i of $\mathcal{S}$ and let C be the completed set of $\mathcal{S}$ after step i. Let C_j^1 be the completed set of $\mathcal{S}_1$ (after step j) that is the largest contained 1-prefix of C. Let P be the set of pebbled nodes after step i of $\mathcal{G}_p$, and let P_j^1 be the set of pebbled nodes after step j of $\mathcal{G}_1$. Note that $P \subseteq C$ and $P_j^1 \subseteq C_j^1$. Let v be an arbitrary node in P. We consider two cases, depending on whether or not v is in C_j^1.

If $v \in C_j^1$, then it has at least one child not in C (otherwise the pebble would have been removed), and hence not in C_j^1 (a subset of C). Since $v \in C_j^1$, it has been pebbled in $\mathcal{G}_1$ at some step no later than j. That pebble could not be removed as yet, since v has a child that has not yet been pebbled as of step j. Thus $v \in P_j^1$. Hence, $P_i \cap C_j^1$ is a subset of P_j.

If $v \notin C_j^1$, then $v \in \mathcal{P}(C) = C - C_j^1$. By Theorem 2.3, the number of premature nodes, $\mathcal{P}(C)$, is at most $(p-1) \cdot (d-1)$. Thus the number of additional pebbled nodes is at worst $(p-1) \cdot (d-1)$.

Considering both cases, we have that $|P| < |P_j^1| + p \cdot d$. Since the number of pebbles used in a pebble game is the maximum of the number of pebbled nodes after each of its steps, we have $Pebbles(G, \mathcal{S}_1) > Pebbles(G, \mathcal{S}) - p \cdot d$. Since $s_1 \geq Pebbles(G, \mathcal{S}_1)$ by definition, the theorem follows. ∎

By Fact 2.1, the number of steps is within a factor of two of optimal.

3.2 The weighted DAG space models

We consider a second class of space models which we call weighted DAG models. In these models nodes are labeled with weights which represent the amount of space that is allocated (positive weight) or deallocated (negative weight) by the action associated with the node. The weighted DAG models differ from the pebble game in that the release of space is modeled directly within the DAG as negative weights rather than as part of the game with the removal of pebbles. This allows us to consider only the transitive reduction of a DAG, which will prove useful for the model of online scheduling discussed in Section 4. We will consider three variants of the model: the first allows for at most unit weights at each node, the second allows for unrestricted weights and the third allows for weights to be associated with groups of nodes, so as to model more flexible allocation and deallocation schemes. In all cases we prove bounds on space (weight) and time (number of steps).

One-weighted DAGs. In our first weighted DAG model, *one-weighted* DAGs, we assume that in addition to a DAG $G = (V, E)$ we have a cost function w that maps the nodes $v \in V$ to integer weights $w(v) \leq 1$. We define the weight (space) of a schedule $\mathcal{S} = V_1, \ldots, V_\tau$ of G as follows. The weight of a completed set $C_j = \cup_{i=1}^j V_i$ is the sum of the weights of its nodes:

$$W(C_j, w) = \sum_{u \in C_j} w(u) \ ,$$

and the weight of the schedule is the maximum weight over its completed sets:

$$W(\mathcal{S}, w) = \max_{j=1,\ldots,\tau} \{W(C_j, w)\} \ .$$

Intuitively, each step of the schedule will add the weights of the newly scheduled nodes to the current accumulated weight, which reflects the current space usage. Since the weights on the nodes can be negative, the accumulated weight can go either up or down from step to step, and it is therefore important to measure the maximum weight (space usage) over all steps.

Given Theorem 2.3, the following space bound is immediate:

Theorem 3.2 (upper bounds for one-weighted DAGs) *Let G be a one-weighted DAG with weight function w and depth $d \geq 1$, and let $\mathcal{S}_1$ be any 1-schedule of G. For all p-schedules $\mathcal{S}$ based on $\mathcal{S}_1$, $W(\mathcal{S}, w) < W(\mathcal{S}_1, w) + p \cdot d$, and if the schedule is greedy then $|\mathcal{S}| < |\mathcal{S}_1|/p + d$.*

Proof. The number of steps $|\mathcal{S}|$ follows from Fact 2.2. By Theorem 2.3, for each step of the p-schedule there are at most $(p-1) \cdot (d-1)$ premature nodes relative to some step of the 1-schedule. Since each node has weight at most 1, there can be at most $(p-1) \cdot (d-1)$ more weight allocated in the p-schedule. ∎

General weighted DAGs. We now consider removing the restriction on the node weights by allowing for integer weights of arbitrary size. Removing this restriction is important in practice since non-constant weights are needed to model the allocation of blocks of space, such as arrays. The difficulty is that each of the premature nodes at a step may add a large weight, thereby exceeding the desired weight bound. For example, if in the DAG used to prove Theorem 2.4, the first half of the nodes in each chain each have weight $k > 1$, the second half of the nodes in each chain each have weight $-k$, and $\mathcal{S}$ is a greedy p-schedule based on a 1DF-schedule $\mathcal{S}_1$, then $W(\mathcal{S}_1, w) = k \cdot d/2$ whereas $W(\mathcal{S}, w) = k \cdot p \cdot d/2$, and hence $W(\mathcal{S}, w) = W(\mathcal{S}_1, w) + \Omega(k \cdot p \cdot d)$, for any k.

Our approach for general weighted DAGs is a straightforward generalization of the one-weighted DAG case. We add to each node v with weight $w(v) > m$, $\lceil w(v)/m \rceil - 1$ dummy nodes with weight 0, where $m \geq 1$ is a tunable parameter. These dummy nodes delay the scheduling of v and therefore decrease the average penalty for each premature node to be at most m, at the cost of an increase in the number of steps. For a given $m \geq 1$ and a weighted DAG $G = (V, E)$, we call the set of nodes $H_m = \{v \in V, w(v) > m\}$ the *heavy* nodes of G, and define the *excess weight* of G as $W_e = W_e(m, w) = \sum_{u \in H_m} w(u)$. We obtain the following result:

Theorem 3.3 (upper bounds for weighted DAGs) *Let G be a weighted DAG with weight function w and depth $d \geq 1$, and let $\mathcal{S}_1$ be any 1-schedule of G. Then for all $p \geq 1$ and all $m \geq 1$, there is a p-schedule $\mathcal{S}$ based on $\mathcal{S}_1$ such that $W(\mathcal{S}, w) < W(\mathcal{S}_1, w) + m \cdot p \cdot d$ and $|\mathcal{S}| < |\mathcal{S}_1|/p + d + W_e/(m \cdot p)$.*

Proof. The proof uses two steps. We will first convert the DAG G into a new DAG G' and the 1-schedule $\mathcal{S}_1$ into a schedule $\mathcal{S}_1'$ on G'. G' will have at most W_e/m more nodes than G. We will then show that a greedy p-schedule of G' based on $\mathcal{S}_1'$ will have the stated bounds.

The graph G' and associated schedule are defined as follows. For each heavy node v of G do the following. In G add $\lceil w(v)/m \rceil - 1$ zero-weight nodes such that each has the same predecessors and successors as v. We call these nodes *dummy* nodes, and they contribute nothing to the computation other than delaying the scheduling of v. In $\mathcal{S}_1$ find the step that contained v and insert $\lceil w(v)/m \rceil - 1$ steps immediately preceding it, each of which schedules one of the dummy nodes. Note that this leads to a valid 1-schedule of the modified DAG since the dummy nodes for v become ready at the same step as v, so they can be scheduled consecutively. Also note that $W(\mathcal{S}_1, w) = W(\mathcal{S}_1', w)$ since the nodes in $\mathcal{S}_1$ get executed in the same order in $\mathcal{S}_1'$, and the additional nodes have zero weight. The total number of dummy nodes in G' will be $\sum_{u \in H_m} \lceil w(u)/m \rceil - 1$, which is at most W_e/m.

Consider the greedy p-schedule of G' based on the 1-schedule $\mathcal{S}_1'$. Note that if a heavy node v is premature at a step of the greedy p-schedule, then all of its dummy nodes must also be premature during that step. This is because they all must be already scheduled (they became ready the same step as v and have a higher priority), and they are all consecutive in $\mathcal{S}_1'$ (if one was not premature, then they would all be not premature). When accounting for the weight of premature nodes, we can therefore amortize the weight of v across all its dummy nodes, since they will always be premature as a group. Since there are $\lceil w(v)/m \rceil - 1$ dummy nodes, the amortized cost is at most m. Since there are at most $(p-1) \cdot (d-1)$ premature nodes, the total weight contributed by premature nodes on any given step is less than $m \cdot p \cdot d$. This gives the desired weight bound. The bound on $\mathcal{S}$ follows from Fact 2.2 since G' has at most $|\mathcal{S}_1| + W_e/m$ nodes, and the schedule is greedy. ∎

Note that if the computation reads and/or writes at least α/m of all the space it allocates, for $1 \leq \alpha \leq m$, then $|\mathcal{S}_1| \geq \alpha W_e/m$, and hence $|\mathcal{S}| < (1 + \frac{1}{\alpha})|\mathcal{S}_1|/p + d$.

Group-weighted DAGs. Our final weighted DAG model associates weights with groups of nodes instead of single nodes, and accounts for the weight of a group when the last of the group is scheduled. This extension is important in practice since memory deallocation can depend on the order of scheduling when using garbage collection. Note that in the pebble game such deallocation can be naturally modeled by the fact that the last child of a node will allow the parent to be unpebbled.

A *group-weighted* DAG is a DAG $G = (V, E)$ and a set R of *groups* each of which contains a subset of the nodes of G and an associated integer weight (*i.e.*, for each group $(g, w) \in R$, $g \subseteq V$ and $w \in \mathcal{I}$). For a schedule $\mathcal{S}$, the weight of the j^{th} completed set is

$$W(C_j, R) = \sum_{(g,w) \in R, g \subseteq C_j} w \ ,$$

and the weight of the schedule is the maximum weight of its completed sets (as before). In this definition the weight of a group is not accounted for until all of its members have been scheduled.

We can prove similar bounds for group-weighted DAGs as for weighted DAGs. Theorem 3.4 below presents upper bounds for group-weighted DAGs with the restriction that all weights are at most 1 and no node belongs to more than one group with a positive weight. We call these *group one-weighted* DAGs. (Extending this result to general group-weighted DAGs is relatively straightforward, and left to the interested reader.)

Theorem 3.4 (upper bounds for group one-weighted DAGs) *Let G be a group one-weighted DAG with groups R and depth $d \geq 1$, and let $\mathcal{S}_1$ be any 1-schedule of G. Then for all p-schedules $\mathcal{S}$ based on $\mathcal{S}_1$, $W(\mathcal{S}, R) < W(\mathcal{S}_1, R) + p \cdot d$, and if the schedule is greedy then $|\mathcal{S}| < |\mathcal{S}_1|/p + d$.*

Proof. The number of steps $|\mathcal{S}|$ follows from Fact 2.2. Consider an arbitrary completed set, C, of $\mathcal{S}$, and let C_j^1 be the completed set of $\mathcal{S}_1$ that is the largest contained 1-prefix of C. Consider any group g with weight w. If $g \subseteq C_j^1$ then w will be accounted for in the weight of both C_j^1 and C, and if $g \not\subseteq C$, then w will not be accounted for in either. In both cases there will be no difference in the weight of the two completed sets due to that group. If $g \subseteq C$ and $g \not\subseteq C_j^1$, then $W(C, R)$ will include w but $W(C_j^1, R)$ will not. Since there are at most $(p-1) \cdot (d-1)$ nodes in $C - C_j^1$ (Theorem 2.3), and each one can account for a weight of at most one, $W(C, R) < W(C_j^1, R) + p \cdot d$. Since this is true for any completed set of $\mathcal{S}$, the weight bound follows. ∎

Matching lower bounds. The following result presents a matching lower bound for all the weighted DAG models we have considered. Since it is a lower bound we need only consider the most restrictive of the models, which are the one-weighted DAGs. We also show that the lower bound is still valid even if we restrict ourselves to series-parallel DAGs.

Theorem 3.5 (lower bounds for weighted DAGs) *For all $p \geq 1$ and $d \geq 3$, there exists a series-parallel one-weighted* DAG *G with weight function w and depth d such that for any greedy p-schedule $\mathcal{S}$ of G, $W(\mathcal{S}, w) = W(\mathcal{S}_1, w) + \Omega(p \cdot d)$, where $\mathcal{S}_1$ is a* 1DF*-schedule. For any p-schedule, the number of steps is at least* $\max\{|\mathcal{S}_1|/p, d\}$.

Proof. Consider the DAG defined in the proof of Theorem 2.5, and let $k = d - 3$. Let w assign weight 0 to the source and sink nodes, weight 1 to all nodes at levels $2, 3, \ldots, d-2$, and weight $-k$ to the nodes at level $d-1$. Then $W(\mathcal{S}_1, w) = k$. Any greedy p-schedule, $\mathcal{S}$, of G must schedule the source node at the first step, all p (newly ready) nodes of level 2 at the second step, and so on. For the completed set $C_{d-2} = \cup_{i=1}^{d-2} V_i$ of $\mathcal{S}$, $W(C_{d-2}, w) = p \cdot k$. It follows that $W(\mathcal{S}, w) = W(\mathcal{S}_1, w) + (p-1) \cdot (d-3)$.

Fact 2.1 gives the lower bound on the number of steps. ∎

Relationship of pebble game and weighted DAGs. We next consider the relationship between the pebble game and weighted DAGs. We show that a pebble game can be transformed into a group-weighted DAG with the same DAG structure so they have identical space requirements. For every node v in a DAG the transformation simply adds a group of weight 1 containing just v, and a group of weight -1 containing the children of v.

Theorem 3.6 (reduction among space models) *For any* DAG *$G = (V, E)$, and for any schedule $\mathcal{S}$ on G,* $\text{Pebbles}(G, \mathcal{S}) = W(\mathcal{S}, R)$*, where the weighted groups R are defined as*

$$R = \{(\{v\}, 1) : v \in V\} \cup \{(\{u : (v, u) \in E\}, -1) : v \in V\} .$$

Proof. We show that at every step V_i of the schedule the net number of pebbles added in the pebble game equals the weight added in the weighted DAG. In the pebble game at step i we will add $|V_i|$ pebbles and remove pebbles from the nodes for which all children become scheduled at the step. In the group-weighted DAG every node belongs to 1 singleton group of weight 1, so these will add $|V_i|$ at the step. Also, any node for which all of its children become scheduled at the step will have a -1 weighted group that becomes contained in this step, therefore contributing a weight of -1. ∎

This shows that the group-weighted DAGs can be used to express any space problem that the p-pebble game can, and are thus equally as expressive. The conversion in the other direction is clearly not possible since a group-weighted DAG can create schedules with negative weights. It is also not possible to create a weighted DAG without groups from a pebble game. We note that it is easy to extend the pebble game to include weighted nodes (with positive weights), and show similar bounds as in Theorem 3.3.

An advantage of a group-weighted DAG is that we need only consider the transitive reduction of the DAG, since the space does not depend on the edges, and the schedule is not effected by transitive edges. This is not possible in the pebble game since transitive edges can change the space requirements. The ability to only consider the transitive reduction of a DAG and not worry about other edges proves helpful in the discussion of online schedules in Section 4.

3.3 Languages and space

In this subsection, we consider how the space models can be used to model a variety of allocation schemes in programming languages: stack allocation, explicit heap management (*e.g.*, `malloc` and `free` in C), and implicit heap management with garbage collection. In all cases, space for the input can be accounted for by placing a set of nodes allocating this space in the first few levels of the DAG (*e.g.*, as roots or as children of a root node). This allows one to model reuse of the input space once it is no longer needed. For both the pebble game and the one-weighted DAG models, we assume each node can allocate any constant amount of memory—this will effect the space bounds by only that constant factor.

The simplest type of allocation is in the form of stack frames. In such allocation, the space for a stack frame is created when entering a function and is used to store local variables, partial results, and control information (*e.g.*, where in the program to return to once the function has completed). The memory for a stack frame is then freed when exiting the function. This is the only type of memory allocation considered by some previous space models [BL93, BL94]. The space required by a stack frame can be modeled in the pebble game by a node representing the entering point for a function, where the stack frame is created, with an edge to a node representing the exiting point of the function, where it is released. In the weighted DAG models, a stack frame can be modeled by a positive weight at the entering node and a corresponding negative weight at the exiting node. In some languages, such as ANSI C (with no extensions), all stack frames are of constant size and can therefore be modeled with constant weight nodes. In other languages, such as Fortran, stack frames can be of dynamic size. In the pebble game, k entering and exiting nodes can be used to model stack frames of size k.

Another standard type of memory allocation is through an explicitly allocated heap, such as with the use of `malloc` and `free` in C. Such allocations/deallocations can be modeled in the pebble game by one or more nodes representing the point where the allocation is executed, the same number of nodes representing where the free is executed, and an edge between each such allocation node and each such deallocation node. Note that if there is no corresponding free, one needs to add dummy nodes at the end of the computation and an edge from each allocation node to each dummy node. In the weighted DAG model, an allocation is modeled with a positive weight and a free by a negative weight. Dummy nodes are not needed, and groups are not required.

We now consider a more interesting use of space, the space required for a language with implicit memory management, such as Lisp, ML, or Java. These languages typically engage a garbage collector to collect freed memory. In the description that follows, we consider how to model the point at which memory becomes available to be reused (*i.e.*, when there is no longer any pointer to the data). This can be thought of as modeling an ideal garbage collector, and allows a model that is independent of the particular garbage collector. The use of implicit heap management differs from explicit heap management in that the node at which a variable becomes free to be collected can depend on the schedule. For example, if there are a set of nodes that reference a variable, then the memory for the variable becomes available for reuse when the last of these nodes is scheduled. (The allocation of memory, however, typically remains independent of the schedule). In the pebble game, we can model such freeing by placing an edge from the allocation node to each of the nodes at which the memory could possibly become available. In the weighted DAG models we need to use group-weighted DAGs—we place a group with the appropriate negative weight around all the nodes at which the memory could become available.

We note that out of the three memory management schemes, groups are only required in the weighted DAGs for the third scheme. Non-constant weights are required whenever a block of memory is allocated of non-constant size, such as in allocating an array proportional to the input size or to an input parameter.

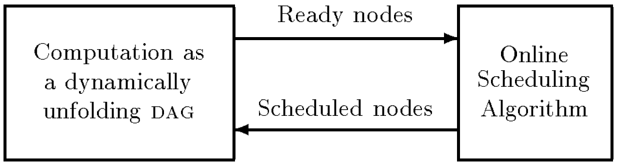
4 Online schedules

The previous sections have shown that any greedy p-schedule based on a 1-schedule is efficient in both space and number of steps. If a DAG and sequential schedule are given as input, this greedy p-schedule is easy to generate. This is called the *offline* scheduling problem. In this section and the next we are concerned instead with the *online* scheduling problem in which neither the DAG nor the sequential schedule is given as input, but rather they are revealed as a computation proceeds. Thus the "list" of tasks in priority order is not given a priori, as in standard list scheduling, but is revealed only as the computation proceeds, and is typically not revealed in order. Our interest in online scheduling is for the implementation of programming languages, where, in general, the structure of a computation is not known a priori.

To execute an online p-schedule based on a 1-schedule there are two capabilities we require. The first is the ability to identify the set of ready nodes, and the second is the ability to keep the current set of ready nodes prioritized according to their 1-schedule. Details on identifying the ready nodes are discussed in the next section, along with a detailed accounting of the cost of scheduling. In this section we give an online algorithm for maintaining the priority of ready nodes relative to their 1DF-numbering, given that the DAG unfolds into a planar graph. The key observation is that it is not necessary to know the full 1DF-schedule to maintain the proper priorities on the ready nodes. These results are of interest in practice since the 1DF-schedule is the most common schedule taken by sequential implementations of languages, and because planar graphs are adequate to model a large class of parallel constructs, including nested parallel loops, parallel divide-and-conquer, and fork-join constructs.

4.1 Online scheduling algorithms

We model computations generated by executing programs as dynamically unfolding DAGs. A dynamically unfolding DAG G is one in which the nodes and edges are revealed as the computation proceeds. Each edge (u, v) in G can be due to either a data or control dependence between the actions associated with nodes u and v (*e.g.*, u spawns v, u writes a value that v reads, v executes conditionally depending on an outcome at u, or v waits for u at a synchronization point). For our space bounds we also assume that a dynamically unfolding DAG is weighted to account for memory that is allocated or deallocated by the actions. As in the offline scheduling problems considered in previous sections, we assume that programs are deterministic, in the sense that a computation (i.e., a particular program run on a particular input data) always unfolds into the same DAG, independent of the schedule. (Extensions to handle nondeterministic programs are discussed in Section 7.)



An *online scheduling algorithm* is the specification of a process that interacts with a dynamically unfolding DAG G. The process maintains a collection of ready nodes R, which is initialized to the roots of G. At each step i the process will specify a new set of nodes (actions) $V_i \subseteq R$ to schedule and will receive a new set of ready nodes. The scheduled nodes are removed from R and the newly ready nodes are added to R. The important property of an online scheduler is that it has no knowledge about the DAG G beyond its previously scheduled nodes, its currently ready nodes, and the fact that it comes from a certain family of DAGs (*e.g.*, series-parallel DAGs, planar DAGs).

4.2 A stack-based scheduling algorithm

We now present a simple online scheduling algorithm for implementing greedy PDF-schedules for DAGs that unfold into planar graphs. The main result used by the algorithm is a theorem showing that, for planar DAGs, the children of any node v have the same (1DF-schedule) priority as v relative to other ready nodes; thus they can be substituted in for v in any sequence of ready nodes ordered by their 1DF-numbers and the sequence remains ordered by 1DF-numbers. This greatly simplifies the task of maintaining the ready nodes in priority order at each scheduling iteration.

We begin by reviewing planar graph terminology. A graph G is *planar* if it can be drawn in the plane so that its edges intersect only at their ends. Such a drawing is called a *planar embedding* of G. A graph $G = (V, E)$ with distinguished nodes s and t is *(s,t)-planar* if $G' = (V, E \cup \{(t,s)\})$ has a planar embedding. In what follows, we will identify various paths in planar embeddings, where we extend our definition of paths to include both the nodes and the directed edges between the nodes.

Recall from Section 2 that a PDF-schedule is based on a given 1DF-schedule. To make a 1DF-schedule unique it is necessary to specify priorities on the outgoing edges of the nodes. Given a planar embedding of a DAG G, we will assume that the outgoing edges of each node are prioritized according to a counterclockwise order, as follows:

Lemma 4.1 *Let G be a DAG with a single root node, s, and a single leaf node, t, such that G is (s,t)-planar, and consider a planar embedding of $G' = (V, E \cup \{(t,s)\})$. For each node v in G', $v \neq t$, let $e_1, e_2, \ldots, e_k$, $k \geq 2$, be the edges counterclockwise around v such that e_1 is an incoming edge and e_k is an outgoing edge. Then for some $1 \leq j < k$, $e_1, \ldots, e_j$ are incoming edges and $e_{j+1}, \ldots, e_k$ are outgoing edges.*

Proof. Suppose there exists an outgoing edge e_x and an incoming edge e_y such that $x < y$. Consider any (directed) path P_1 from the root node s to node v whose last edge is e_1, and any (directed) path P_y from s to v whose last edge is e_y. Let u be the highest level node that is on both P_1 and P_y but is not v. Let C be the union of the nodes and edges in P_1 from u to v, inclusive, and in P_y from u to v, inclusive. Then C partitions G into three sets: the nodes and edges inside C in the planar embedding, the nodes and edges outside C in the planar embedding, and the nodes and edges of C.

Note that one of e_x or e_k is inside C and the other is outside C. Since $v \neq t$, t must be either inside or outside C. Suppose t is outside C, and consider any path P from v to t that begins with whichever edge e_x or e_k is inside C. P cannot contain a node in C other than v (since G is acyclic) and cannot cross C (since we have a planar embedding), so other than v, P contains only nodes and edges inside C, and hence cannot contain t, a contradiction. Likewise, if t is inside C, then a contradiction is obtained by considering any path from v to t that begins with whichever edge e_x or e_k is outside C.

Thus no such pair, e_x and e_y, exist and the lemma is proved. ∎

Let G be an (s,t)-planar DAG with a single root node s and a single leaf node t. We say that G has *counterclockwise edge priorities* if there is a planar embedding of $G' = (V, E \cup \{(t,s)\})$ such that for each node $v \in V$, the priority on the outgoing edges of v (used for the 1DF-schedule of G) is according to a counterclockwise order from any of the incoming edges of v in the embedding (*i.e.*, the priority order for node v in the statement of Lemma 4.1 is $e_{j+1}, \ldots, e_k$). Thus the DAG is not only planar, but the edge priorities at each node (which can be determined online) correspond to a planar embedding. Such DAGs account for a large class of parallel languages including all nested-parallel languages, as well as other languages such as Cilk.

Maintaining priority order for planar graphs. We now present a simple algorithm that, for any valid (parallel) schedule of a planar DAG G, maintains the set of ready nodes at each step sorted by their 1DF-numbers.

ALGORITHM Maintain-Order:
R is an ordered set of ready nodes initialized to the root of G. Repeat until R is empty:

1. Schedule any subset of the nodes from R.
2. Replace each newly scheduled node with its zero or more ready children, in priority order, in place in the ordered set R. If a ready child has more than one newly scheduled parent, consider it to be a child of its lowest priority parent in R.

Theorem 4.2 below shows that for planar DAGs, the children of any node v have the same (1DF-schedule) priority as v relative to other ready nodes; hence the Maintain-Order algorithm maintains the set of ready nodes R in priority order.

Theorem 4.2 *For any single root s, single leaf t, (s,t)-planar DAG G with counterclockwise edge priorities, the online Maintain-Order algorithm maintains the set R of ready nodes sorted by their 1DF-numbers.*

Proof. We first prove properties about the 1DF-numbering of G, and then use these properties to argue that the Maintain-Order algorithm maintains the ready nodes in relative order of their 1DF-numbers.

Let $G = (V, E)$, and consider the planar embedding of $G' = (V, E \cup \{(t,s)\})$ used to define the counterclockwise edge priorities. We define the *last parent tree* for the 1DF-schedule of G to be the set of all nodes in G and, for every node $v \neq s$, we have an edge (u,v) where u is the parent of v with highest 1DF-number. Note that a 1DF-schedule on the last parent tree would schedule nodes in the same order as the 1DF-schedule on G.

Consider any node u that is neither s nor t. Define the "rightmost" path $P_r(u)$ from s to u to be the path from s to u in the last parent tree. Define the "leftmost" path $P_l(u)$ from u to t to be the path taken by always following the highest-priority child in G. Define the *splitting path* $P_s(u)$ to be the path obtained by appending $P_r(u)$ with $P_l(u)$.

In the embedding, the nodes and edges of the cycle $P_s(u) \cup \{(t,s)\}$ partition the nodes not in $P_s(u)$ into two regions — inside the cycle and outside the cycle — with no edges between nodes in different regions. Consider the counterclockwise sweep that determines edge priorities, starting at any node in the cycle. If the cycle is itself directed counterclockwise (clockwise), this sweep will give priority first to any edges in the outside (inside, respectively) region, then to edges in the cycle, and then to any edges in the inside (outside, respectively) region. A node w not in $P_s(u)$ is *left* of $P_s(u)$ if it is in the region given first priority; otherwise it is *right* of $P_s(u)$.

We claim that all nodes left (right) of $P_s(u)$ have 1DF-numbers less than (greater than, respectively) u. The proof is by induction on the level in G of the node. The base case, $\ell = 1$, is trivial since s is the only node at level 1. Assume the claim is true for all nodes at levels less than ℓ, for $\ell \geq 2$. We will show the claim holds for all nodes at level ℓ.

Consider a node w at level ℓ, and let x be its parent in the last parent tree; x is at a level less than ℓ. Suppose w is left of $P_s(u)$. Since there are no edges between left and right nodes, x is either in $P_s(u)$ or left of $P_s(u)$. If x is in $P_s(u)$ then (x,w) has higher priority than the edge in $P_s(u)$ out of x. Thus by the definition of $P_l(u)$, x cannot be in $P_l(u)$. If x is in $P_r(u)$, then a 1DF-schedule on the last parent tree would schedule x and w before scheduling any more nodes in $P_s(u)$ (including u). If x is left of $P_s(u)$, then u is not a descendant x in the last parent tree (since otherwise x would be in $P_r(u)$). By the inductive assumption, a 1DF-schedule on the last parent tree would schedule x before u, and hence schedule any descendant of x in the last parent tree (including w) before u. Thus w has a 1DF-number less than u.

Now suppose w is right of $P_s(u)$. Its parent x is either right of $P_s(u)$ or in $P_s(u)$. If x is right of $P_s(u)$, then by the inductive assumption, x and hence w has a 1DF-number greater than u. If w is a descendant of u, then w has a 1DF-number greater than u. So consider $x \neq u$ in $P_r(u)$. A 1DF-schedule on the last parent tree will schedule the child, y, of x in $P_r(u)$ and its descendants in the tree (including u) before scheduling w, since (x,y) has higher priority than (x,w). Thus w has a 1DF-number greater than u.

The claim follows by induction.

Now consider a step of the Maintain-Order algorithm and assume that its ready nodes R are ordered by their 1DF-numbering (lowest first). We want to show that a step of the algorithm will maintain the ordering. Consider two nodes u and v from R such that u has a higher priority (*i.e*, a lower 1DF-number) than v. Assume we are scheduling u (and possibly v). Since both u and v are ready, u cannot be in the splitting path $P_s(v)$. Since u has a lower 1DF-number than v, it follows from the claim above that u is left of $P_s(v)$. Since there are no edges between nodes left and right of a splitting path, the children of u are either in $P_s(v)$ or left of $P_s(v)$. If a child is in $P_s(v)$ then it is a descendant of v and the child would not become ready without v also being scheduled. But if v were scheduled, u would not be the lowest priority parent of the child, and hence the algorithm would not assign the child to u. If a child is to the left of $P_s(v)$, then by the claim above, it will have a lower 1DF-number than v. When placed in the position of u, the child will maintain the 1DF-number ordering relative to v (and any children of v) in R. Likewise, for any node w in R with higher priority than u, w and the children of w (if w is scheduled) will have lower 1DF-numbers than u and its children. Since the Maintain-Order algorithm schedules a subset of R and puts ready children back in place, it maintains the nodes in R sorted by their 1DF-numbers. ∎

Implementing a greedy PDF-schedule. We now present our algorithm for implementing a greedy PDF-schedule of a planar DAG. The algorithm is simply a restriction on the Maintain-Order algorithm to greedily schedule the highest priority nodes from R at each step.

ALGORITHM P-stack:
R is an ordered set of ready nodes initialized to the root of G. Repeat until R is empty:

1. Schedule the first $\min\{p, |R|\}$ nodes from R.
2. Replace each newly scheduled node with its zero or more ready children, in priority order, in place in the ordered set R. If a ready child has more than one newly scheduled parent, consider it to be a child of its lowest priority parent in R.

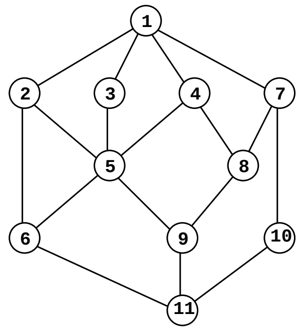
Figure 6 shows an example of the P-stack algorithm.

Corollary 4.3 *For any single root, s, single leaf, t, (s,t)-planar* DAG *G with counterclockwise edge priorities, the online P-stack algorithm produces the greedy* PDF*-schedule based on the* 1DF*-schedule of G.*

Proof. Since the algorithm schedules a subset of R and puts ready children back in place, it maintains R ordered relative to the 1DF-numbering (Theorem 4.2). Since it executes a greedy p-schedule based on selecting the nodes with lowest 1DF-numbers, it executes the greedy PDF-schedule based on the 1DF-schedule. ∎

5 Machine implementations

In this section, we consider a concrete implementation of the online greedy PDF-schedule approach outlined in the previous section, and analyze its performance on a variety of parallel machine models. The primary tasks not addressed in the previous section are how to identify ready nodes, which requires coordination among the parents of a node, and how to manage the P-stack data structure within the desired bounds. Our time and space bounds include all scheduling, synchronization and computation costs for implementing a nested-parallel language. The bounds, however, do not consider the time costs for memory allocation/deallocation, since these costs depend on the type of memory management, which is highly language-dependent. We briefly address memory management costs in Section 7, where we show that certain types of memory management schemes can also be implemented within our time bounds.



$R_1 = \{1\}$	$V_1 = \{1\}$
$R_2 = \{2,3,4,7\}$	$V_2 = \{2,3\}$
$R_3 = \{4,7\}$	$V_3 = \{4,7\}$
$R_4 = \{5,8,10\}$	$V_4 = \{5,8\}$
$R_5 = \{6,9,10\}$	$V_5 = \{6,9\}$
$R_6 = \{10\}$	$V_6 = \{10\}$
$R_7 = \{11\}$	$V_7 = \{11\}$

Figure 6: An example of the P-stack algorithm on a planar DAG with $p = 2$. The numbers on the nodes represent the 1DF-numbering. For $i = 1, \ldots, 7$, R_i is the ordered set of ready nodes at step i and V_i is the set of scheduled nodes at step i.

We limit ourselves to the implementation of languages that support nested-parallel constructs. A *nested-parallel* construct is one in which a process can spawn (fork) a set of parallel child processes. The spawning process (*i.e.*, the parent) is then suspended until all the child processes finish. The child processes can themselves spawn their own child processes, allowing for a nesting of the parallelism. The descendants of a process cannot have dependences among each other. We assume there is an ordering among the child processes, and that a sequential implementation will execute a 1DF-schedule of the DAG based on this ordering. It is not hard to show that the transitive reduction of the DAG resulting from executing a program in any nested-parallel language will always be a (single source and sink) series-parallel DAG (as defined in Section 2). This has been shown for the NESL language [BG96]. Any series-parallel DAG G is (s,t)-planar with counterclockwise edge priorities, where s is the single root source node and t is the single leaf sink node. Thus the P-stack algorithm of Section 4.2 can be used to implement a PDF-schedule on the DAG.

Nested-parallel constructs include parallel loops, parallel maps, and fork-join constructs, and any nesting of these constructs. All nested data-parallel languages, such as Paralation-Lisp [Sab88], Proteus [MNP+90], or NESL [BCH+94] are nested-parallel. Other nested-parallel languages include PREF-ACE [Ber88] and PTRAN [ABC+88]. Languages such as PCF [Lea89] and HPC++ [BGJ96] are nested as long as locks, or other synchronization mechanisms, are not used among processes. The language Cilk [BJK+95] is not strictly nested, but can be converted into a nested form by only increasing the work and depth by a small constant factor.

We consider three implementations of nested-parallel languages. The first implementation assumes that each node of the DAG has at most binary fanout. Such DAGs are sufficient for languages that allow each process to spawn only a constant number of child processes per step. The second allows for unbounded fanout but assumes the underlying machine has a fetch-and-increment operation (defined below). Allowing for unbounded fanout makes it possible to represent many computations with asymptotically less depth than when restricted to binary fanout. The third implementation we consider shows how unbounded fanout can be implemented on an EREW PRAM model using a plus-scan (all-prefix-sums) operation instead of a fetch-and-increment operation.

Definitions and terminology. We begin by defining the terms we use in this section. A process has *started* if its first node (action) has been scheduled, and has *finished* if its last node has been scheduled. A process is *live* if it has started but not finished. A live process is *active* if one of its nodes (actions) is ready and *suspended* otherwise (*i.e.*, when waiting for one of its children to complete). A process is *ready* if its first node is ready but has not been scheduled. A *spawn node* is the node (action) at which a process spawns a set of children, and a *synchronization node* is the node at which the process restarts after the children have finished. All the processes that are spawned by the same spawn instruction are called *siblings*.

We say a nested-parallel computation uses *concurrent access primitives* if two or more nodes that are unordered by its DAG read or write the same memory location; if there are no such nodes, the computation uses *exclusive access primitives*.

To model space in nested-parallel languages we will use the weighted DAG models, and we therefore need only consider the transitive reduction of the computation DAGs, which are series-parallel. In the implementation of any language, each process needs storage space (a "frame") for local data, a program code pointer, and a pointer to the frame of its parent process (analogous to a return pointer in serial languages). We will call these *process frames* and assume the space for them is accounted for on the appropriate nodes of the weighted DAG (*e.g.*, a positive weight at the beginning of the process and a negative weight at the end). In addition to the process frames, our space model allows memory to be allocated anywhere in a process and deallocated anywhere in a (perhaps different) process as long as there is a path in the DAG from the allocating node to the deallocating node.

The implementations we present employ data structures for scheduling whose space is proportional to the number of live processes. We will therefore use the following bound on live processes to bound the additional space used by the scheduler:

Lemma 5.1 (bound on live processes) *If a PDF-schedule is used to schedule a nested-parallel computation of depth $d \geq 1$ on $p \geq 1$ processors, there can be at most $p \cdot d$ live processes at any step.*

Proof. Note that a 1DF-schedule can have at most $d' \leq d$ live processes at any step, where d' is the depth of the node scheduled that step. This is because only one sibling will be live at a time (a previous sibling will always finish before the next starts). For a PDF-schedule we can have at most $(p-1)\cdot(d-1)$ premature nodes (Theorem 2.3), each of which can start at most one additional process. Thus the number of live processes is at most $p \cdot d$. ∎

We will use three operations in our implementations: a fetch-and-increment, a plus-scan and a copy-scan. In a *fetch-and-increment* operation, a processor can fetch a value stored at an address, and then increment the value at that address by 1. The fetch and the increment occur together as a single atomic operation, such that if a set of k processors fetch-and-increment to the same location concurrently then it will appear as if the k fetch-and-increments occurred in some sequential order. A set of n fetch-and-increment operations (possibly to different locations) can be implemented work-efficiently (*i.e.*, in $O(n)$ work) in $O(\lg n)$ time on a concurrent-read, concurrent-write (CRCW) PRAM, and can be implemented as efficiently as concurrent reads or writes on certain networks [Ran91, LMRR94]. However, there is no known polylogarithmic time work-efficient implementation on an EREW PRAM, and fetch-and-increment is generally considered an expensive operation in practice. For the plus-scan and copy-scan operations, it is assumed that there is a total order on the processors. In a *plus-scan* operation (also called all-prefix sums) each processor has an integer input value and receives as output the sum of the values of preceding processors. In a *copy-scan* operation each processor has an integer input value and an input flag and receives as output the value of the nearest preceding processor whose flag is set. The plus-scan and copy-scan operations for n virtual processors can be implemented work-efficiently on an EREW PRAM with a very simple algorithm in $O(\lg n)$ time, and can be implemented directly in hardware with a simple tree.

5.1 Binary fanout

The implementation of an online scheduler needs to be able to identify the ready nodes and maintain the ordering of the ready nodes in the P-stack algorithm. Here we discuss how this can be achieved when each process can spawn at most two new processes on a given step.

In the series-parallel DAGs resulting from nested-parallel computations, ready nodes can either be the children after a spawn instruction, the node following a normal instruction in a process, or a synchronization node. The first two cases are easy to identify so we consider how to identify when a synchronization node becomes ready, *i.e.*, when the last child finishes so that we can restart the computation of the parent. Since the computation is unfolding dynamically, the scheduler does not know ahead of time which child will finish last or how long any of the child computations will take. Furthermore the scheduler cannot afford to keep the parent active (busy-waiting) since this could lead to work-inefficiency (recall that such spawning can be nested). We will call this problem of identifying when a synchronization node is ready the *task-synchronization* problem.

In the case of binary spawning the task-synchronization problem is relatively easy to handle. When a process is started we allocate two flags, a *synch-flag* and a *left-right flag*. Whenever a process does a spawn, it sets its synch-flag to 0 and sets the left-right flags of its two children to 0 and 1 respectively. The idea of the synch-flag is that it will be set to 1 by the first of the two children to finish. In particular, when a process finishes it checks its parent synch-flag. If the synch-flag is 0 the process sets it to 1. If the synch-flag is 1 the other branch has finished and the process identifies the parent process as ready. To avoid having the two children check the flag simultaneously we can have the children check on alternate instruction cycles, *e.g.*, left children (those with left-right flag set to 0) check on odd cycles and right children (those with left-right flag set to 1) check on even cycles.[4] The test only requires constant time and can be executed with exclusive access (EREW).

We now consider how the online scheduler maintains the P-stack data structure. The scheduler stores the stack as an array in which each element is a pointer to a process frame (or code to initialize a process frame) and the two flags mentioned above. The elements of the array are therefore of constant size. At each step we take the first p actions off of the stack and execute them each on their own processor (the i^{th} processor picks the i^{th} action off the stack). By definition each action takes constant time. The actions can result in either 0, 1 or 2 new ready actions (0 if a process finishes and the parent is not ready, 1 if continuing to the next step in a process or if a process finishes and the parent is ready, and 2 if executing a spawn). A plus-scan operation on these counts is used to determine for each processor how many lower priority new actions exist before its new actions. These results are then used to write the new ready actions back into the P-stack, ahead of any actions that were not taken off the stack this step. As long as the plus-scan is executed across the processors in the same order as they were taken from the stack, then the scheduler will maintain the P-stack in the proper order for the P-stack algorithm, and will therefore execute a PDF-schedule of the computation DAG. All the above steps can be executed in constant time on a p-processor EREW PRAM augmented with a plus-scan primitive, and hence in $O(\lg p)$ time on a $(p/\lg p)$-processor standard EREW PRAM.

Theorem 5.2 (implementation with binary spawning) *Consider a computation expressed in a nested-parallel language with binary spawning, which does w work, has depth d, requires sequential space s_1 for the group one-weighted space model, and uses (only) exclusive access primitives. This computation can be implemented by the online scheduler described above in $O(w/p + d)$ time and $s_1 + O(p \cdot d)$ space on a p-processor* EREW PRAM *augmented with a unit-time, linear-work plus-scan primitive, including all costs for computation, synchronization and scheduling.*

Proof. We have discussed how task-synchronization and each step of the scheduler can be implemented in constant time. We have also shown that the scheduler will execute a PDF-schedule, and therefore by Theorem 3.4 that the space of the computation is bound by $s_1 + O(p \cdot d)$. This however does not include the space for the P-stack nor the additional flags we added, since these are part of the scheduler rather than the computation itself. We can account for every element in the P-stack by charging it to its parent, which must be live, and we note that due to the binary spawning restriction, any process can have at most two children in the P-stack at a time. Since there can be at most $p \cdot d$ live processes at any given time (Lemma 5.1), and each allocates just two flags and accounts for at most two elements in the P-stack, the total space for the scheduler is bounded by $O(p \cdot d)$. ∎

[4] A test-and-set instruction could also be used.

The construction used in Theorem 3.3 to allow for arbitrary weights does not directly apply to binary spawning since it assumes that all the dummy nodes have the same parent and child. It is not hard to extend it to allow for binary spawning, however, by creating a tree of dummy nodes, but this might increase the depth of the computation. Since in the following sections we will allow for unbounded spawning, we will defer to these sections any further considerations of non-constant allocations.

5.2 Unbounded fanout with fetch-and-increment

We now extend the previous results to allow for spawning of an arbitrary number of processes on each step. Allowing for unbounded spawning makes it more complicated to deal with both scheduling the processes and synchronizing them when they finish. In this section we allow for a fetch-and-increment instruction which makes the task-synchronization problem relatively straightforward, so that the main concern becomes how to maintain the schedule within the stated space bounds. In the next section we consider the more complicated case of synchronizing without a fetch-and-increment.

To implement task-synchronization we use a counter. When a process is started, instead of allocating two flags we allocate two counters and an integer field. The *start-counter* will be used by the scheduler and is described below. The *end-counter* is used after the process does a spawn, and it counts how many of its child processes have finished. The *num-spawned* integer field is used to hold the number of child processes spawned by a process. Whenever a process does a spawn, it sets both counters to 0 and sets num-spawned to be the number of spawned child processes. Whenever a process finishes, it increments its parent's end-counter using a fetch-and-increment. If the increment brings the counter up to num-spawned, then the finishing child process identifies its parent as ready. Since the fetch-and-increment is atomic, only one of the siblings will identify the parent as ready, and only after all others have finished.

We now consider how to maintain the P-stack data structure. The problem with the scheduler as described in Section 5.1 is that when allowing for unbounded spawning we can no longer bound the number of ready nodes in the P-stack by $O(p \cdot d)$. Instead what we do is store the ready nodes in a compressed form. When a process does a spawn, instead of adding all the spawned children to the P-stack it adds a *stub* entry that can be used to create the children lazily. This stub entry includes the num-spawned and the start-counter. Whenever child processes are taken from the stub (started) the counter is incremented, and when the counter reaches num-spawned, the stub is removed. Each stub entry therefore represents the ready, but not started, child processes of a spawn instruction. This is similar to the idea of control blocks used by Hummel and Schonberg to implement parallel loops [HS91]. To allocate actions to processors at a step, we expand out the first p actions from this compressed representation of the P-stack. Using standard processor allocation techniques, this can be accomplished with a plus-scan and copy-scan while maintaining the order among the actions [Ble89]. As before, each processor executes an action. If it has an action from a stub, it executes the first step of the corresponding child process thereby starting that process. Each action will result in either no new action (a finishing node), a single new action, or a new stub entry (a spawning node). These can be placed back into the P-stack in the correct order using a plus-scan operation.

We omit any formal proof that this approach maintains the bounds stated in Theorem 5.2, since we prove a stronger theorem in the next section, but it is not hard to see that only $O(p \cdot d)$ space is required by the scheduler since each premature node can add at most constant space to the compressed representation of the P-stack. Furthermore, since we allow for unbounded spawning, we can allow for the allocation of arbitrary-sized blocks at each node and use Theorem 3.3 to bound the time and space.

5.3 Unbounded fanout without fetch-and-increment

We now consider how to solve the task-synchronization problem with unbounded spawning without using a fetch-and-increment instruction. The problem is that without a fetch-and-increment we have no easy way to increment the end-counter in parallel in order to detect when the last child completes. One possible solution is to build a binary tree when the processes are spawned which will be used to synchronize as they finish. If n threads are spawned, this involves a $\lg n$ slowdown to go up the tree when synchronizing, and unless dynamic load balancing is used, will also require extra work. Such an implementation therefore loses the advantage of allowing for arbitrary fanout—the simulation costs equal the extra depth in the DAG required by binary fanout.

Description of the algorithm and data structures. We avoid using a fetch-and-increment or a binary tree for task-synchronization using the following approach:

1. We generate a coordination list among siblings when they are spawned.

2. As each child finishes, it removes itself from the list by short-cutting between its two neighbors. If neither neighbor is finishing on the same step, the short-cutting takes constant time.

3. If multiple adjacent neighbors finish, we use a copy-scan computation to shortcut over all completing neighbors. To make this possible, we use properties of the 1DF-schedule to show that all neighbors that are completing will be adjacent in the P-stack. Note that neighbors that are not completing might not be in the P-stack at all since they might be suspended (*i.e.*, they might have live children).

4. When the last child finishes, it reactivates the parent. If multiple finish simultaneously, then the leftmost reactivates the parent.

This approach allows us to simulate each step of the PDF-schedule in constant time on a p-processor EREW PRAM augmented with copy-scan and plus-scan primitives. The remainder of this subsection presents the details.

A coordination linked list. Whenever we start a process we allocate a link-node for it with previous and next pointers which are used to maintain a bidirectional *coordination list*. We will maintain the invariant that the previous pointer points to the link-node of the previous live process among its siblings and the next pointer points to the next live or ready process among its siblings. If there is no such process the pointer is set to a special value NULL. A node is the only unfinished sibling if both of its pointers are NULL. This makes it easy to recognize when to reactivate the parent. We deallocate each link-node when its process finishes.

Maintaining the coordination list. The coordination list must be maintained under the addition of more live siblings as they are started and under the deletion of finishing siblings. We first consider how to create the coordination list. As in Section 5.2, we will use stub entries in the P-stack to represent a set of ready child processes. In each stub entry we will maintain a last-child pointer to the link-node of the last child process that was started (the one with the lowest priority). The last-child pointer is initialized to NULL. On any step we can start a sequence of processes (siblings) from a stub entry, and create a link node for each. For the first of these processes we set its previous-pointer to the last-child pointer. For the last of these processes, we set its next-pointer to NULL if the stub has been emptied, and otherwise we set its next-pointer to point to the stub entry and set the last-child pointer of the stub to point to its link-node. The remaining sibling processes are cross-linked among each other. This is easy to do since they will be in adjacent processors.

We now consider maintaining the coordination list under deletions. If a single sibling finishes, it can simply splice itself out of the list. The possible difficulty is that a sequence of up to p consecutive nodes in the list can be deleted at the same time. Standard parallel pointer jumping (*i.e.*, list-ranking or chaining) can be used to update the links among the remaining nodes, but this may be rather slow (*e.g.*, the best known algorithm on an EREW PRAM augmented with copy-scan and plus-scan primitives takes $\Theta(\lg p)$ time), and work-efficient algorithms may be quite involved [Rei93]. Instead, we use the special features of our data structure to derive a fast and simple solution for updating the links. The key observation, presented in Lemma 5.3, is that if a sequence of two or more adjacent sibling nodes is deleted, then their representatives reside in consecutive entries of the P-stack (despite any further spawning that may have occurred). Hence, updating the coordination list for this deleted sublist can be implemented by executing a copy-scan to copy the appropriate pointers across the consecutive entries.

Lemma 5.3 *If a sequence of two or more adjacent sibling nodes is deleted from the coordination list at a step, then the finishing nodes of these sibling processes are represented in consecutive entries of the P-stack.*

Proof. First, we note the following four facts. (i) Sibling nodes are put in their coordination list in the order of their 1DF-numbers. (ii) Nodes in the P-stack are ordered by their 1DF-numbers. (iii) A node may only be deleted from the coordination list if its associated finishing node is in the P-stack. (iv) When a finishing node is placed in the P-stack, it is ready, and hence its ancestors can neither be in the P-stack or subsequently placed in the P-stack.

Next, the reader may verify the following property of DF-schedules in series-parallel DAGs: Given $k \geq 2$ consecutive sibling processes, with starting nodes $x_1, \ldots, x_k$ and finishing nodes $y_1, \ldots, y_k$, then the only nodes with 1DF-numbers greater than y_1's 1DF-number but no more than y_k's 1DF-number are the nodes that are, for some $i = 2, \ldots, k$, either (a) x_i, (b) y_i for $i < k$, or (c) both a descendant of x_i and an ancestor of y_i.

Finally, let u and v be adjacent nodes in the current coordination list, u before v, such that both are to be deleted at this step. By fact (iii), their finishing nodes are both in the P-stack. For every sibling node w between u and v among the siblings, since w is no longer in the coordination list, w has already been deleted, and hence by facts (iii) and (iv), no ancestor of w's finishing node can be in the P-stack. Similarly, since v is in the P-stack, no ancestor of its finishing node can be in the P-stack. The lemma follows by facts (i) and (ii), and the above property. ∎

Finally, we note that when all remaining siblings finish on a step, the leftmost of these is responsible for identifying the parent as ready.

Large allocations. As discussed in Section 3.2, large allocations can be implemented within the space bounds by placing dummy nodes into the computation. In a nested-parallel computation these can easily be added by having an allocation of a non-constant size n spawn a set of n/m (for any constant m) dummy processes which finish immediately. The actual allocation is made at the synchronization node.

Complexity. Each step of the PDF-schedule involves at most p unit-time actions. The operations on the data structures in the scheduling algorithm described above for one step of a PDF-schedule can be implemented in a constant number of steps on a p-processor EREW PRAM plus a constant number of applications of the plus-scan and copy-scan operations. We state the following theorem for an EREW PRAM augmented with copy-scan and plus-scan primitives. Since the scan operations can be implemented work-efficiently on a standard EREW PRAM in $O(\lg p)$ time, this implies bounds of $O(w/p + d \lg p)$ time and $s_1 + O(p \cdot d \cdot \lg p)$ space for a standard EREW PRAM. The same bounds also apply with high probability on a hypercube using standard simulations of the PRAM and standard implementations of scans on a hypercube [Val90]. The following theorem is valid for any of the weighted DAG models.

Theorem 5.4 (nested-parallel implementation) *Consider a nested-parallel computation with work w, depth d, sequential space s_1, which allocates at most $O(w)$ space, and uses (only) exclusive access primitives. This computation can be implemented by the online scheduler described above in $O(w/p + d)$ time and $s_1 + O(p \cdot d)$ space on a p-processor* EREW PRAM *augmented with unit-time, linear-work copy-scan and plus-scan primitives, including all costs for computation, synchronization and scheduling.*

Proof. We first note that the transformation to deal with large allocations will effect the work and depth of the computation by at most a constant factor. The work is only effected by a constant factor since we are assuming the computation allocates at most $O(w)$ space. By Corollary 4.3, the P-stack algorithm described will execute a PDF-schedule. By Theorem 3.3 using a constant for m, there are $O(w/p + d)$ steps in the PDF-schedule and the space for the computation will be within the specified bounds. As discussed above, all steps can be executed in constant time. We now consider the space for the scheduler, which is limited to the space for the P-stack and for the link-nodes. We note that only live processes can contribute an entry to the P-stack and each live process can contribute at most one entry, either one of its actions or a stub entry for a set of its ready children. Furthermore the link-node is only needed while a process is live. Since there are at most $p \cdot d$ live processes at any time (Lemma 5.1), and the stub entries and link-nodes are of constant size, the space required by the scheduler is bound by $O(p \cdot d)$. ∎

For the CRCW PRAM (without any extra primitives), the time bounds may be improved over those obtained for the standard EREW PRAM by replacing the copy-scan and plus-scan operations with randomized CRCW algorithms for chaining and approximate prefix-sums. The details are left to the interested reader.

6 Related work

In this section, we discuss previous related works on parallel scheduling, including early work on time bounds for parallel schedules, work on heuristics to limit memory usage, work on scheduling algorithms with provable space bounds, work on scheduling for languages with nested parallelism, work on automatic processor allocation, and work on pebble games.

In early parallel scheduling work, Graham [Gra66, Gra69] modeled a parallel computation as a DAG of tasks, where each task takes an arbitrary, but known number of time units and must be scheduled on a single processor without preemption. He introduced the notion of list-schedules in which there is a total order among the tasks that dictates the scheduling priorities, and proved that a particular greedy list schedule is within a factor of two of optimal There was a large body of research that extended this work in various ways, much of which is summarized by Coffman [Cof76]. Brent [Bre74] showed that a DAG with n unit cost nodes, and depth d can be scheduled on p processors in less than $n/p + d$ steps using a breadth-first schedule. Some more recent work has considered issues of communication costs (*e.g.* [PU87, PY88]). Burton et al. [BMRS90] considered modeling the execution of parallel languages as dynamically unfolding DAGs and showed how some of the previous work on static DAGs could be applied. None of the above work considered memory usage.

Several papers have developed scheduling heuristics for limiting the space required by functional/dataflow programs (*e.g.* [BS81, Hal85, RS87, CA88]). An early work that provided provable space bounds for tree-structured programs was due to Burton [Bur88].

Blumofe and Leiserson [BL93, BL94] considered space and time bounds for a multithreaded model in which each thread contains any number of tasks. Their results also related the parallel space and time to the work w of a program, the critical path length or depth d of a program, and the space s_1 required to execute it sequentially. They showed that using their model and scheduling scheme the parallel space is bounded by $O(s_1 \cdot p)$ and the parallel time is bounded by $O(w/p + d)$, where p is the number of processors. Their model differs from ours in several respects. First, they assume that all data needs to be stored within the threads. When a new thread is spawned, memory is allocated for the entire thread and deallocated only when the thread terminates. In our model data can be viewed as residing in a global pool, and we allow individual tasks to allocate and deallocate data on-the-fly. Another difference is that each task in our model can spawn an arbitrary number of new tasks at a time, whereas each thread in their model can spawn only one new thread at a time. Because of this, their work-stealing algorithm [BL94] is likely not appropriate in

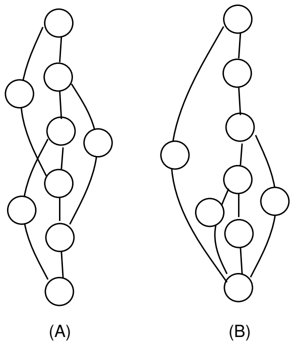


Figure 7: Two DAGs corresponding to fully-strict computations. The nodes down the center of both DAGs correspond to a single parent thread.

our model, and they do not solve the general processor allocation and task-synchronization problems we address. An advantage of their model is that since the memory is associated with the threads it is easier to account for communication costs, and in fact they prove good bounds on the total amount of communication needed by the computation under their scheduling algorithm.

Blumofe and Leiserson prove their bounds for fully-strict computations, which are similar but not identical to nested-parallel computations. In a fully-strict computation, a thread need not suspend when it forks a child. It can synchronize at any point with its parent, but with no other thread. Figure 7(a) illustrates a strict computation that is neither purely nested nor planar. It is not hard, however, to modify our online algorithm to work with fully-strict computations. Furthermore, the actual synchronizations used in Cilk [BJK+95] (the language in which their bounds are applied) only allow for synchronization points that join all children, as shown in Figure 7(b). Such synchronizations lead to planar graphs that can be scheduled by our Maintain-Order algorithm.

In more recent work, Blumofe et al. [BFJ+96] have shown improved bounds for their work-stealing scheduler when applied to regular divide-and-conquer algorithms (*i.e.*, algorithms that divide the data into a equal-sized subproblems each of size n/b). For example for a divide-and-conquer matrix multiply, where $a = 8$, $b = 2$ and the sequential space, s_1, is n^2, their new space bounds for p processors are $s_p = \Theta(p(n/p^{1/\lg_b a})^2) = \Theta(n^2 p^{1/3})$. These bounds are not as good as the $s_p \leq n^2 + O(p \lg^2 n)$ bounds obtained by our techniques.

Burton and Simpson [BS94] (see also [Bur96]) developed a scheduling algorithm for dynamically unfolding DAGs. In their most general model, associated with each node is the time to perform the task and the amount of memory allocated or deallocated by the task. Nodes in the DAG may have arbitrary fanin and fanout. This roughly corresponds to our weighted DAG model except that their tasks are not necessarily unit-time tasks. Their model does not permit schedule-dependent deallocation, and hence cannot model languages using garbage collection. They present a scheduling algorithm for p processors that guarantees at most $s_1 \cdot p$ space and at most $w/p + d$ steps (ignoring all scheduling overheads), where the sequential space s_1 is the worst case space bound considering all possible depth-first schedules. In contrast, we define the sequential space to be the space used by the standard 1DF-schedule, *i.e.* the 1DF-schedule in which the leftmost ready child is explored. Scheduling overheads are considered for DAGs with unit-time tasks and constant fanout. For such DAGs, they present a decentralized, randomized work-stealing algorithm that guarantees $O(s_1 \cdot p)$ space and whose expected time is within a constant factor of optimal for programs with sufficient parallelism. This time bound ignores any overheads associated with identifying ready nodes (in particular, detecting when a task with multiple parents becomes ready, *e.g.*, the task-synchronization problem we solve).

Various techniques have been considered for implementing nested parallel languages [FTYZ90, HS91]. A scheduling technique with provable time bounds was presented in [Ble90]. The time bounds are the same as presented in this paper on the same model and include all scheduling and synchronization costs, but the results are limited to a subclass of nested-parallel programs called *contained* programs. The technique was the basis for the original implementation of the NESL programming language [BCH+94]. Similar results where shown by Suciu and Tannen for a parallel language based on while loops and map recursion [ST94]. Blelloch and Greiner [BG95] proved that the nested-parallelism available in call-by-value functional languages can be mapped onto the CREW PRAM with scan primitives with the same time bounds as given in this paper. This parallelism is limited to binary fanout. None of the above work considered space bounds.

Several recent works have studied automatic processor allocation in certain computation paradigms, such as task decaying algorithms, geometric decaying algorithms, spawning algorithms, and loosely-specified algorithms; the proposed scheduling techniques were typically based on very fast CRCW PRAM algorithms for relaxed versions of the prefix-sums problem such as linear compaction, load balancing, interval allocation, and approximate prefix-sums [GM91, GMV91, Goo91, Hag91, MV91, Mat92, Hag93, Gil94, GM96, GMV94, GZ95]. The techniques that were used are insufficient, however, to cope with the model considered in this paper, even when space considerations are ignored. In particular, previous techniques assumed that whenever a thread goes to sleep, it is known precisely which step it will awake. Thus the task-synchronization problem we solve does not arise in these previous models.

There have been many papers on sequential pebble games, most of which have studied various space-time tradeoffs. Pippenger gives a good summary of most of the early work [Pip80]. The parallel pebble game we use was described by Savage and Vitter [SV84]. They showed that an n-input FFT graph can be p-pebbled in $O(n^2/(sp) + (n \lg s)/p)$ time using $\lg n \leq s \leq O(n/\lg n)$ pebbles. This result allows for repebbling. They, however, did not show any general results bounding pebbles and time for arbitrary graphs. Two player parallel pebble games have been considered [VT85, VT89] and used to characterize various parallel complexity classes, such as LOGCFL and AC^1. These games seem unrelated to the parallel pebble game we consider.

7 Discussion

This paper has presented important new results for offline and online scheduling, including the first bounds on the space of greedy parallel schedules that are within a factor of

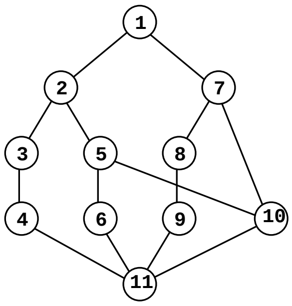


Figure 8: A non-planar embedding of a DAG for which the P-stack algorithm will fail to execute a PDF-schedule for $p = 3$. The nodes are labeled with their 1DF-numbers. The P-stack algorithm will generate the schedule $\{1\}, \{2, 7\}, \{3, 5, 8\}, \{4, 6, 10\}, \{9\}, \{11\}$ and therefore schedule nodes 9 and 10 out of order.

$1 + o(1)$ of the sequential space (assuming sufficient parallelism). We conclude the paper by discussing various recent extensions of this work and some areas for future research.

Nondeterminism. This paper has been concerned with deterministic programs, in which the computation DAG is independent of the order in which nodes are scheduled. Some of the results can be extended to nondeterministic programs, *e.g.*, programs with race conditions, although then the bounds are based on the worst-case DAGs over all schedules, as follows. Nondeterministic programs can be viewed as programs whose computation is a nondeterministic selection from a set, $\mathcal{G}$, of deterministic DAGs. Consider a deterministic procedure for generating a 1-pebbling of an arbitrary DAG. Then a p-pebbling based on the 1-pebbling procedure uses at most $\max_{G \in \mathcal{G}}(P_1(G)+d(G) \cdot p)$ pebbles, where $P_1(G)$ is the number of pebbles in a 1-pebbling of the DAG G and $d(G)$ is the depth of G.

Futures. Although the P-stack algorithm works for all (s,t)-planar DAGs with counterclockwise edge priorities, it is not guaranteed to work for more general classes of DAGs. Figure 8 gives an example DAG on which the given edge priorities (as revealed by the 1DF-numbers) result in a non-planar embedding, and the P-stack algorithm will not generate a PDF-schedule. This sort of DAG can appear when using futures [BH77, Hal85], and the P-stack algorithm therefore will not in general execute a PDF-schedule in languages that support futures. The work in this paper has been extended to generate the PDF-schedule for such languages by using a 2-3 tree data structure that maintains the ready set in the appropriate priority order [BGMN97].

Reducing scheduling overheads. Our schedules assume that each node represents a unit-time action. If these actions are just a single operation such as an addition, then the constant-time cost for scheduling the node could significantly outweigh the cost of executing the action. To avoid this it is important to schedule larger blocks of work as a single node. If we assume memory is allocated at the beginning or end of a block then grouping computations into fixed-size blocks does not alter our space bounds. This, however, would require that a compiler break up threads into fixed-sized blocks so they can be preempted at the end of each block. Following up the results in this paper, Narlikar and Blelloch [NB97] presented a scheduling algorithm that runs jobs mostly nonpreemptively, while maintaining the same space bounds as presented in this paper. The basic idea is to allocate a fixed pool of memory to a thread when it starts and then allow it to run nonpreemptively until it either terminates, forks new threads, or runs out of memory from its pool. They also presented experimental results demonstrating that the technique is both time and space efficient in practice.

Reducing communication costs. Our scheduling algorithm does not consider machine locality. The technique discussed in the previous paragraph can help keep locality and reduce communication costs since it will keep threads on the same processor as long as possible. However, provable bounds on communication, such as those presented by Blumofe and Leiserson [BL94], have not been shown, and since the technique tends to schedule the child threads on different processors than the parent, it is unlikely to be as communication-efficient as the approach in [BL94]. An open question is whether any communication bounds can be shown for scheduling algorithms related to the one discussed in this paper, or possibly for hybrid algorithms that combine ideas from this paper and from [BL94].

Memory allocation procedures. The space bounds given earlier account for the absolute number of memory cells used, without addressing the issue of explicit memory allocation for our data structures and for the program variables declared during the execution. Memory allocation for maintaining the P-stack as an array is straightforward. Memory allocation for maintaining the link-nodes for each process can be done using a dynamic dictionary data structure. For the program variables themselves allocation will depend on the management scheme. If we assume all memory is allocated as stack frames or using allocate and free (*i.e.* no garbage collection) then the program memory can also be maintained using a dynamic dictionary data structure. Hence, an adaptive allocation and deallocation of space, so as to maintain at each step space which is linear in the number of live processes, or in the number of program variables, can be implemented in logarithmic time on a p-processor EREW PRAM [PVW83], and in $O(\lg^* p)$ time and linear work with high probability, on a CRCW PRAM [GMV91].

Acknowledgements. The authors thank the anonymous referees for comments that helped improve the presentation of this paper.

References

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *J. of Parallel and Distributed Computing*, 5(5):617–40, October 1988.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, October 1989.

[BCH+94] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *J. of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[Ber88] D. Bernstein. PREFACE-2: Supporting nested parallelism in Fortran. Technical Report Research Report RC-14160, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.

[BFJ+96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[BG95] G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Proc. 7th International Conf. on Functional Programming Languages and Computer Architecture*, pages 226–237, June 1995.

[BG96] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proc. ACM SIGPLAN International Conf. on Functional Programming*, pages 213–225, May 1996.

[BGJ96] P. Beckman, D. Gannon, and E. Johnson. Portable parallel programming in HPC++. In *Proc. 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 132–139, August 1996.

[BGMN97] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 12–23, June 1997.

[BH77] H. G. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, August 1977.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiseron, K. H. Randall, and Y. Zhou. CILK: An efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[BL93] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 362–371, May 1993.

[BL94] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pages 356–368, November 1994.

[Ble89] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, C-38(11):1526–1538, 1989.

[Ble90] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, Cambridge, MA, 1990.

[BMRS90] F.W. Burton, G.P. McKeown, and V.J. Rayward-Smith. Applications of uet scheduling theory to the implementation of declarative languages. *The Computer Journal*, 33(4):330–336, 1990.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. of the ACM*, 21(2):201–208, 1974.

[BS81] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pages 187–194, October 1981.

[BS94] F. W. Burton and D. J. Simpson. Space efficient execution of deterministic parallel programs. Manuscript., December 1994.

[Bur88] F. W. Burton. Storage management in virtual tree machines. *IEEE Trans. on Computers*, 37(3):321–328, 1988.

[Bur96] F. W. Burton. Guaranteeing good memory bounds for parallel programs. *IEEE Trans. on Software Engineering*, 22(10):762–773, 1996.

[CA88] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proc. 15th International Symp. on Computer Architecture*, pages 141–150, May 1988.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, NY, 1990.

[Cof76] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Son, New York, 1976.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *J. of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[FTYZ90] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Trans. on Computers*, 39(7):919–929, 1990.

[Gil94] J. Gil. Renaming and dispersing: Techniques for fast load balancing. *J. of Parallel and Distributed Computing*, 23(2):149–157, November 1994.

[GM91] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, January 1991.

[GM96] J. Gil and Y. Matias. An effective load balancing policy for geometric decaying algorithms. *J. of Parallel and Distributed Computing*, 36(2):185–188, August 1996.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[GMV94] M.T. Goodrich, Y. Matias, and U. Vishkin. Optimal parallel approximation algorithms for prefix sums and integer sorting. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 241–250, January 1994.

[Goo91] M.T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 711–722, 1991.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. of Applied Mathematics*, 17(2):416–429, 1969.

[GZ95] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Proc. 3rd Israel Symp. on Theory of Computing and Systems*, pages 220–228, January 1995.

[Hag91] T. Hagerup. Fast parallel space allocation, estimation and integer sorting. Technical Report 03/91, SFB 124, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Germany, 1991.

[Hag93] T. Hagerup. Fast deterministic processor allocation. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 1–10, 1993.

[Hal85] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[HPV77] J. Hopcroft, W. Paul, and L. Valiant. On time versus space. *J. of the ACM*, 24(2):332–337, April 1977.

[HS91] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM J. of Research and Development*, 35(5/6):743–765, 1991.

[JP92] S. Jagannathan and J. Philbin. A foundation for an efficient multi-threaded Scheme system. In *Proc. 1992 ACM Conf. on Lisp and Functional Programming*, pages 345–357, June 1992.

[Lea89] B. Leasure. PCF programming model and FORTRAN bindings. In *Proc. 13th International Computer Software and Applications Conf.*, page 111, September 1989.

[LMRR94] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *J. of Algorithms*, 17(1):157–205, July 1994.

[Mat92] Y. Matias. *Highly Parallel Randomized Algorithmics*. PhD thesis, Tel Aviv University, Israel, 1992.

[MNP+90] P. H. Mills, L. S. Nyland, J. F. Prins, J. H. Reif, and R. A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report UNC-CH TR90-041, Computer Science Dept., University of North Carolina, 1990.

[MV91] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

[NB97] G. J. Narlikar and G. E. Blelloch. Space-efficient implementation of nested parallelism. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 25–36, June 1997.

[Pip80] N. Pippenger. Pebbling. In *Proc. 5th IBM Symp. on Mathematical Foundations of Computer Science: Computational Complexity*, May 1980.

[PU87] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM J. on Computing*, 16(4):639–646, 1987.

[PVW83] W.J. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proc. 10th Int. Colloquium on Automata Languages and Programming, Springer LNCS 154*, pages 597–609, 1983.

[PY88] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 510–513, May 1988.

[Ran91] A. G. Ranade. How to emulate shared memory. *J. of Computer and System Sciences*, 42:307–326, 1991.

[Rei93] J. H. Reif, editor. *A Synthesis of Parallel Algorithms*. Morgan-Kaufmann, San Mateo, CA, 1993.

[RS87] C. A. Ruggiero and J. Sargeant. Control of parallelism in the Manchester dataflow machine. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, Vol. 174*, pages 1–15. Springer-Verlag, 1987.

[Sab88] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, Massachusetts, 1988.

[ST94] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 57–66, June 1994.

[SV84] J. Savage and J. Vitter. Parallelism in space-time tradeoffs. In *Proc. International Workshop on Parallel Computing and VLSI*, pages 49–58, May 1984.

[Val90] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 943–972. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[VT85] P. W. Venkateswaran and M. Tompa. Speedups of deterministic machines by synchronous parallel machines. *J. of Computer and System Sciences*, 30:149–161, 1985.

[VT89] H. Venkateswaran and M. Tompa. A new pebble game that characterizes parallel complexity classes. *SIAM J. on Computing*, 18(3):533–549, 1989.