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
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
gibbons@research.att.com

Yossi Matias
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
matias@research.att.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 dynamically schedule tasks to processors so as to not only minimize the execution time, but also to minimize the amount of memory needed. Without careful scheduling, the parallel execution on p processors can use a factor of p or larger more memory 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, even for programs whose task structure is revealed only during execution. For programs with sufficient parallelism, the schedule guarantees that the amount of memory used by the program is within a factor of $1+o(1)$ of a sequential implementation. This space bound is obtained by proving a graph-theoretic result relating parallel and sequential traversals of directed acyclic graphs.

The paper then describes an efficient dynamic scheduling algorithm that generates schedules in this class, for languages with *nested* fine-grained parallelism. The algorithm is relatively simple, performing the necessary processor allocation and task synchronization while incurring at most a constant factor overhead in time and space. The correctness and performance guarantees of the algorithm rely on properties of depth-first-like traversals of series-parallel graphs. The algorithm is the first efficient solution to the scheduling problem discussed here, even if space considerations are ignored.

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 dependencies (the *depth*). For example, consider the following pseudo-code for multiplying two 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 used [BS81, Hal85, RS87, CA88], but these are not guaranteed to be space efficient in general.

In this paper we are interested in specifying universal implementations that guarantee performance bounds, both in terms of time and space. These are specified by placing upper bounds on the running time and the space of the implementation as a function of the work, depth and sequential space. As with the work of Blumofe and Leiserson [BL93, BL94], we formalize the notion of work, depth and space, by modeling computations as directed acyclic graphs (DAGs) that may unfold dynamically as the computation proceeds. The nodes in the DAG represent unit-work tasks, and the edges represent any ordering dependencies between the tasks that must be respected by the implementation. This is a very general model, modeling even pro-

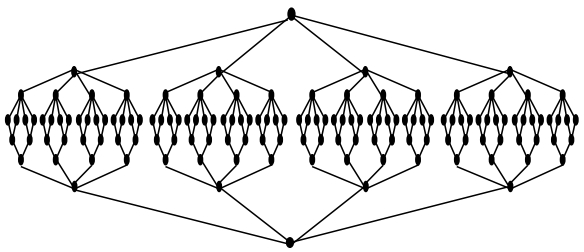

Figure 1: The task structure of a matrix multiplication computation (for $n = 4$) represented as a directed acyclic graph. Nodes represent unit-work tasks, and edges (assumed to be directed downward in the figure) represent control and/or data flow between the tasks. A level-by-level schedule of this 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.

grams (such as for parallel quicksort [JáJ92, Ble95]) whose task structure or data dependencies are revealed only as the execution proceeds. 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. To account for memory usage, we assign a weight to each node of the DAG that represents the amount of memory that node needs to allocate or deallocate. The sequential space of a computation is then defined as the input space plus the space required by the depth-first traversal of its DAG (the traversal taken by standard sequential implementations).

Any parallel schedule that makes good use of the processors will almost always schedule tasks in a different order than the sequential implementation. This can result in an increase in both the memory needed for task bookkeeping (to keep track of perhaps a larger set of ready tasks at each step) and the amount of memory allocated to program variables (to hold a possibly larger set of variables that have been allocated but not yet deallocated). Note that while the task bookkeeping space may be proportional to the number of nodes on the "frontier" of the schedule at a step, the program variable space is the *sum of the weights* of *all* the nodes scheduled prior to the step. As a result, even a parallel schedule that maintains a small frontier can observe a large space increase, and it may be somewhat surprising that universally good bounds can be obtained.

1.1 Provably efficient schedules

We first identify a class of parallel schedules that are provably efficient in both space and number of steps, for any dynamically unfolding DAG. If a computation has work W and depth D, and takes S_1 sequential space, then a p-processor schedule from this class has the following performance guarantees:

- There are at most $W/p + D$ steps in the schedule. This is always within a factor of two of the best possible over all schedules. For programs with sufficient parallelism (*i.e.* $W/p \gg D$), this is within a factor of $1 + o(1)$ of the best possible.
- The computation uses only $S_1 + O(p \cdot D)$ space. This includes space for program variables and for task bookkeeping. Thus for programs with sufficient parallelism (*i.e.* $S_1/p \gg D$, recalling that S_1 is at least the size of the input), this is within a factor of $1 + o(1)$ of the sequential space. Previously, the best known bound was $S_1 \cdot p$ [Bur88, BL93, BL94, BS94], a factor of p from the sequential space.

These bounds apply when individual tasks allocate at most a constant amount of memory. When unit-work tasks can allocate memory in arbitrary amounts, the same space bound can be obtained using at most $(W + S_1)/p + D$ steps, and in general we have a trade-off between space and steps. The space bound implies, for example, that the memory required for matrix multiplication is $S_1 + O(p \lg n)$, which is within a factor of $1 + o(1)$ of the best possible for all $p \in o(n^2/\lg n)$.

The above bounds do not account for overheads to implement the schedule.

Approach and key ideas. To obtain a good bound on the number of steps, we use the common approach (*e.g.* [BL93]) of greedily scheduling p independent nodes of the DAG each step, if possible, where p is the number of processors. To obtain our space bounds, we focus on bounding the increase in memory for program variables, since our solution suffices as well to bound the increase in memory for task bookkeeping. By labeling *individual* nodes with their memory requirements, we allow for more fine-grained memory allocation than in previous models that associate memory requirements with entire threads in the computation [BL93, BL94]. Block memory allocations, *e.g.* for arrays, are indicated by nodes whose weight is the size of the block to be allocated.

The primary question is which (greedy) parallel schedules, if any, have provably good space bounds on all computation DAGs. Our first key idea is to 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 on each step. Our second key idea is to focus on these "premature" nodes, and show with a careful argument that the number of premature nodes at any step of the p-processor schedule is at most p times the depth, D, of the DAG. This matches a lower bound we show and justifies the use of parallel schedules based on sequential schedules.

Our third key idea is to use this bound on the number of premature nodes to bound the space requirements. At each parallel step of the computation, each premature node may require space beyond that needed for the sequential schedule, due to additional allocations performed by these nodes and in order to keep track of any additional nodes ready to be scheduled. We show an approach for ensuring that the extra space for premature nodes is linear in their number. Since there are at most $p \cdot D$ premature nodes, we obtain our $S_1 + O(p \cdot D)$ space bound, where S_1 is the space for the sequential schedule.

The above results apply to any parallel schedule based on a sequential schedule. Since the standard sequential schedule is a depth-first traversal (DFT) of the DAG, we define and consider p-DFT schedules, a class of parallel schedules based on sequential depth-first schedules. There are a number of ways one might think to define a parallel DFT; we show that our definition has provably good performance bounds.

1.2 An efficient scheduling algorithm

The p-DFT class of parallel schedules we define are provably efficient in both time and space relative to the standard sequential depth-first traversal. Our second main result is an efficient runtime (online) scheduling algorithm for generating p-DFT schedules, for languages with *nested* fine-grained parallelism (*i.e.* languages that lead to series-parallel DAGs). We show how the processor allocation and task synchronization needed to implement p-DFT schedules can be performed by the processors on-the-fly with minimal overhead. For a computation with W work, D depth, and S_1 sequential space, the resulting scheduled computation, including these overheads, obtains the following bounds. We obtain $O(W/p+D\cdot\lg p)$ time on an EREW PRAM, and the same time bound, with high probability, on a hypercube, for EREW-style computations (*i.e.* no concurrent reads or writes). On the stronger CRCW PRAM, a variant of the algorithm obtains $O(W/p+D\cdot\lg\lg p)$ time worst case or $O(W/p+D\cdot\lg^* p)$ time with high probability, for CRCW-style computations (*i.e.* concurrent reads and writes permitted at no extra cost). These work-efficient schedules use only $\Theta(S_1 + D\cdot p\lg p)$ or better space. If tasks can allocate memory in arbitrary amounts, the same bounds can be obtained as long as the sequential space is at worst linear in the sequential running time (*i.e.* $S_1 \in O(W)$).

These results apply to nearly all *data-parallel* languages, both nested and not, as well as most languages that supply fork-and-join style parallelism, even permitting arbitrary fanout and arbitrary nesting. In the appendix we show how the NESL [Ble93, BCH+94] language, a nested data-parallel language, maps onto the appropriate type of DAGs. Together with previous and recent results on automatic memory allocation [GMV91, BGM95], our algorithm can be used for time-, work-, and space-efficient execution of programs written in these languages. The algorithm is the first efficient solution to the scheduling problem discussed here, even if space considerations are ignored.

Approach and key ideas. First, we observe that for the class of languages we consider, the task structure of a computation is a dynamically unfolding series-parallel DAG, with certain nodes having arbitrary fanout and other, corresponding nodes having arbitrary fanin. Next, we show that for such DAGs, a simple stack-based parallel scheduling algorithm yields a p-DFT schedule, as well as specifying how the processors should be allocated to the tasks. Thus a parallel schedule based on a sequential one can be constructed on-the-fly without knowing the sequential schedule.

Our simple stack-based algorithm requires considerably more than $O(S_1 + p\cdot D)$ stack space for some DAGs, so a modified algorithm, using lazy stack allocation, is given that obtains this bound on its stack space. Moreover, given the unknown and varying levels of recursive nesting in nested parallelism computations and the possibly large fanin at a node, there are difficulties in efficiently identifying nodes that become available for scheduling. Our next key idea is an efficient algorithm for this *task-synchronization* problem. We use properties of p-DFT schedules on series-parallel DAGs to argue the correctness of our algorithm and its data structures, as well as its resource bounds.

Our stack-based scheduling algorithm performs a constant number of parallel prefix-sums operations and a constant number of EREW PRAM steps for each round of scheduling. To amortize these scheduling overheads, we schedule multiple tasks per processor at each round by performing a parallel DFT whose scheduling width is larger than the number of processors. This increases the additive term in the time and space complexities, but ensures that the parallel work is within a constant factor of optimal. We also show how a near-greedy scheduling algorithm using approximate prefix-sums can be used to improve the time complexity for the more powerful CRCW PRAM model.

Outline.
Section 2 discusses related work, and the computation model is defined in Section 3. Section 4 presents our results for provably efficient schedules, while Section 5 presents our scheduling algorithm. Further discussion is in Section 6, and the appendix describes the mapping of NESL to our computation model.

2 Related work

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. Time bounds are proven under various scenarios, including a result that a particular greedy schedule is within a factor of two of optimal. Brent [Bre74] showed that a level-by-level schedule of a DAG obtains a time bound within a factor of two of optimal. There have been several previous works on scheduling static DAGs (*e.g.* [PU87, PY88]). Memory usage was not considered in any of this work. An early work that provided provable space bounds for tree-structured programs was due to Burton [Bur88].

Blumofe and Leiserson [BL93, BL94] considered space, time and communication bounds for a multithreaded model in which each thread contains any number of tasks. They showed that with their model and scheduling scheme the parallel space is bounded by $O(S_1\cdot p)$ and parallel time is bounded by $O(W/p+D)$. We extend their model by allowing for global shared memory, dynamic memory allocation, and arbitrary fanin and fanout of the DAGs. These extensions allow us to cover a significantly wider class of applications and languages, but makes it much harder for us to account for communication costs.

Recently, and independently, Burton and Simpson [BS94] developed a scheduling algorithm for dynamically unfolding DAGs. Their model, like ours, allows for dynamic memory allocation and for arbitrary fanin and fanout but differs in that it defines S_1 as the memory used in the worst case of all possible depth-first traversals. In contrast, we define S_1 to be the space used by the standard DFT, *i.e.* the DFT in which the leftmost ready child is explored. Also, as with the work of Blumofe and Leiserson, their space bounds are $O(S_1\cdot p)$ rather than our $S_1 + O(p\cdot D)$.

Provable time bounds for mapping nested data-parallel languages onto the PRAM were considered in [Ble90]. These results were used for implementing the NESL language [Ble93] but the time bounds are restricted to a class of programs that are called *contained*. With this implementation, a contained NESL program with W work and D depth runs in $O(W/p+D\lg p)$ time on a p-processor PRAM. 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 PRAM 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, and the emphasis of the work is more on the full mapping from a language and

less on the scheduling.

Several recent works have studied automatic processor allocation in certain computation paradigms, such as task decaying algorithms, geometric decaying algorithms, and loosely-specified algorithms (see [GMV91, ?, ?, Mat92, MV91] and references therein). The techniques that were used, however, do not consider space bounds and are insufficient to cope with the model considered in this paper, since they 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.

3 Computations as dynamically unfolding DAGs

In this section, we introduce the terminology used in this paper to describe the computation DAGs and their schedules, including the general model we use to measure the space required to execute a program according to a given schedule of the DAG.

Graph terminology. We use standard graph terminology (see, *e.g.*, [CLR90]), some of which is reviewed below. Consider a directed acyclic graph G. A *p-traversal* (or *p-schedule*) of G, for $p \geq 1$, 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 three properties hold: (i) each node appears exactly once in the schedule; (ii) a node is scheduled only after all its ancestors have been scheduled in previous steps; and (iii) each step consists of at most p nodes. A *traversal* (or *schedule*) of G is a p-traversal (or p-schedule) of G for some p.

Consider a traversal $T = V_1, \ldots, V_\tau$ of G. A node $v \in G$ is *scheduled* prior to a step i in T if v appears in $V_1 \cup \cdots \cup V_{i-1}$. An unscheduled node v is *ready* at step i in T if all its ancestors (equivalently, all its parents) are scheduled prior to step i. The *frontier* of G at step i in T is the set of all nodes scheduled prior to i with an unscheduled child node. A *greedy p-traversal* [BL93] is a p-traversal 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.

A *depth-first* 1-traversal (DFT or 1-DFT) is obtained by maintaining a stack of ready nodes: the stack contains the root nodes initially (in any order), and at each step, the top node on the stack is popped from the stack and scheduled, and then any newly ready nodes are pushed on the front of the stack (in any order). The ith node in a DFT is said to have DFT *number* i.

A (single source and sink) *series-parallel* DAG is 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 $i \neq j$.

Parallel schedules based on sequential ones. We define a p-traversal, T_p, to be *based on* a 1-traversal, T_1, if, at each step i of T_p, the k_i earliest nodes in T_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 T_1, then either both are scheduled, neither are scheduled, or only u is scheduled. Note that given a 1-traversal, the greedy p-traversal based on the 1-traversal is uniquely defined.

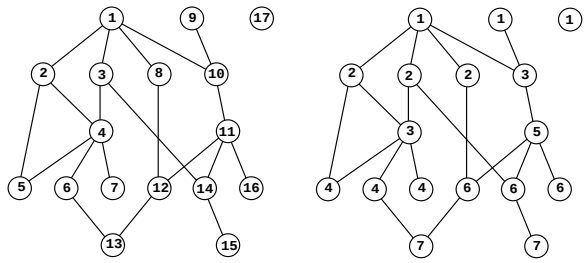

Figure 2: A greedy p-DFT of a DAG G, for $p = 3$. On the left, the nodes of G are numbered in order of a 1-DFT of G. On the right, G is labeled according to the greedy p-DFT, T_p, based on the 1-DFT: $T_p = V_1, \ldots, V_7$, where for $i = 1, \ldots, 7$, V_i, the set of nodes scheduled in step i, is the set of nodes labeled i in the figure.

An important p-traversal that we consider in Section 5 is the depth-first p-traversal. A *depth-first p-traversal* (p-DFT) is a p-traversal based on a depth-first 1-traversal. An example is given in Figure 2.

In general, implementing a p-traversal based on a 1-traversal requires some method of determining the relative order in the 1-traversal among ready nodes at each step of the p-traversal.

Dynamically unfolding DAGs. We model a computation as a DAG that unfolds dynamically as the program executes on a given input. As in previous work (*e.g.* [PY88, BL93, BL94]), we assume the programs are deterministic, in the sense that the DAG for a computation does not depend on the order in which nodes are scheduled. There is a node in the DAG for each unit-work task in the computation, which we identify with the task. The edges represent any ordering dependencies between the tasks—if the program dictates that u must be executed before v, then there is a path in the DAG from u to v. Such ordering could be due to either data or control dependencies. A node may have arbitrary indegree and outdegree. The program is assumed to define an ordering on the edges outgoing from a node. We note that our DAGs differ from dataflow graphs since a dataflow edge from a node u to v need not be included in the DAG if there is another path from u to v—the DAG strictly represents ordering constraints and not the flow of data.

The DAG is *dynamically unfolding* in the following sense: (i) when a node is scheduled, its outgoing edges are revealed; and (ii) when all its incoming edges are revealed, a node is revealed and is available for scheduling. In an *online* scheduling algorithm for a dynamically unfolding DAG, the scheduling decision at each step is based on only the revealed nodes and edges of the DAG. Initially, only the root nodes are revealed, and the algorithm must detect when new nodes are revealed in order to schedule them.

The depth of the DAG is the parallel depth, D, of the computation; the number of nodes in the DAG is the total number of unit-work tasks, W, of the computation. Since the programs are deterministic, D and W are not effected by the traversal order. Note that a deterministic program may still be based on a randomized algorithm; in such cases the DAG may depend on the values of the random bits, which may be viewed as part of the input data for the program.

In the full paper, we extend some of our results to non-deterministic programs, *e.g.* programs with race conditions, although then the bounds we obtain are based on worst case DAGs over all traversals.

Space model. We consider two categories of space: (i) *program variable space* defined by tasks when they are scheduled, including space for task representations (the "stack frames") as well as any dynamic memory use, and (ii) *task bookkeeping space* used by the scheduling algorithm to keep track of nodes ready to be scheduled.

Program variable space is the memory to hold the input, the memory for stack frames, the memory explicitly allocated by program instructions, and the memory implicitly allocated to hold values computed by the program. The input space is assumed to reside in a preallocated block of memory; the remainder of the variable space is allocated by individual tasks. We assume that the amount of memory to be allocated by a task is independent of the traversal of the DAG. For deallocation, such an assumption is overly restrictive for languages that rely on garbage collection to automatically deallocate memory for values that are no longer needed. In particular, the memory for a value is ready to be deallocated as soon as the last task that references the value has completed. Thus certain deallocations are associated with a set of tasks, *i.e.* those tasks that reference the value, such that the last such task to be scheduled is credited for the deallocation.

At any point in the computation, the program variable space in use is the input space plus the sum of all the space allocated by scheduled tasks minus the sum of all the space deallocated by scheduled tasks. We can assign a weight, $w(u)$, to each task u that represents the amount of space allocated by the task minus the amount deallocated. For a prefix $T = V_1, \ldots, V_j$ of a p-traversal, $p \geq 1$, we define $\text{Space}(T)$, the program variable space in use after T to be $\text{Space}(T) = n + \sum_{i=1}^{j} \sum_{u \in V_i} w(u)$, where n is the amount of space needed to hold the input. This definition assumes a common pool of memory so that any deallocated space can be re-used by subsequent allocations. Moreover, by considering only the net effect on the space of all tasks scheduled at a step, it ignores the fluctuations in memory use during the step as tasks allocate and deallocate. Such fluctuations can be addressed by splitting each node into two, one that allocates and one that deallocates, if desired. The *space complexity* or *maximum space* of a p-traversal, $T_p = V_1, \ldots, V_\tau$, is defined as $S_p = \max_{j=1,\ldots,\tau} \text{Space}(V_1, \cdots, V_j)$, *i.e.* the maximum space in use after any step of the traversal.

For task bookkeeping, we assume that a node in the DAG that is identified with a task u is of constant size, and also that each edge is constant size. Consider the sequence, S, of (revealed) edges outgoing from a node. Any consecutive subsequence of S can be represented compactly in constant space by storing the first and last edge in the subsequence. However, an edge must be allocated constant storage of its own before it can be used to identify the node at its other endpoint. Bookkeeping space for a node must be allocated before the node can be scheduled, and can be deallocated after it is scheduled. Note that although the scheduling algorithm can base its decisions on all revealed nodes and edges of the DAG at a step, it need not store all these nodes and edges. In fact, it need only store sufficiently many nodes and edges to be able to identify and schedule ready nodes within the desired resource bounds for the scheduling algorithm.

We claim that the assumptions made in defining our computation model with its space model are reasonable for most fine-grained languages. For example, the model accurately reflects the execution of a NESL program.

4 Efficient space bounds for general DAG traversals

Consider a dynamically unfolding DAG, G, with n nodes and depth d, such that individual tasks allocate at most a constant amount of memory. In this section, we prove matching upper and lower bounds on the space overheads of greedy parallel schedules relative to sequential schedules. In particular, we show that for any sequential traversal of G, and any number of processors $p \geq 2$, the greedy p-traversal of G based on the sequential traversal uses only $O(p \cdot d)$ more space than the sequential traversal. Our main lemma shows that for any prefix, σ_p, of a p-traversal, there is a prefix, σ_1, of the sequential schedule such that σ_p contains all the nodes of σ_1 and at most $p \cdot d$ additional nodes. Since each of these additional nodes can allocate only constant space, our space bound will follow. Extensions to handle allocations of arbitrary size are discussed at the end of this section.

Our upper bounds on space hold for any p-traversal based on a 1-traversal, greedy or not. An advantage of focusing on *greedy* p-traversals is that they are provably efficient in their number of steps. Blumofe and Leiserson, generalizing a result due to Brent [Bre74], showed the following:

Theorem 4.1 ([BL93]) *For any* DAG *of n nodes and depth d, and for any $p \geq 1$, the number of parallel steps in any greedy p-traversal T_p is at most $n/p + d$.*

This bound on the number of steps accounts only for the computation as defined by the nodes of the DAG; it does not account for overheads in allocating processes to nodes or for any bookkeeping overheads in implementing the program. Likewise, in this section we do not consider time and space overheads for processor or space allocation procedures. Furthermore, we assume that the p-traversal can determine the relative order in the 1-traversal among ready nodes at a step, with no overhead. Such overheads are addressed in Section 5, where a complete scheduling algorithm is presented for p-DFTs on DAGs arising from languages with nested fine-grained parallelism.

4.1 Tight bounds on premature nodes in p-traversals

For almost all DAGs, a p-traversal that schedules more than one node per step will schedule nodes out-of-order with respect to the 1-traversal on which it is based.

Let T_p be a p-traversal based on a 1-traversal T_1. For each prefix, σ_p, of T_p, consider the longest prefix, σ_1, of T_1 that includes only nodes in σ_p. We say a node is *premature with respect to* σ_p if it is in σ_p but not in σ_1. Let $\text{P}(\sigma_p)$ be the set of premature nodes with respect to σ_p. Figure 3 shows the premature nodes with respect to two steps of the p-DFT of Figure 2. The *maximum number of premature nodes* in T_p is $\max\{|\text{P}(\sigma_p)| : \sigma_p \text{ is a prefix of } T_p\}$.

The following lemma gives an upper bound on the maximum number of premature nodes.

Lemma 4.2 (upper bound on premature nodes) *For any* DAG *of depth d, and for any 1-traversal T, the maximum number of premature nodes in any p-traversal based T is at most $(p-1) \cdot (d-1)$.*

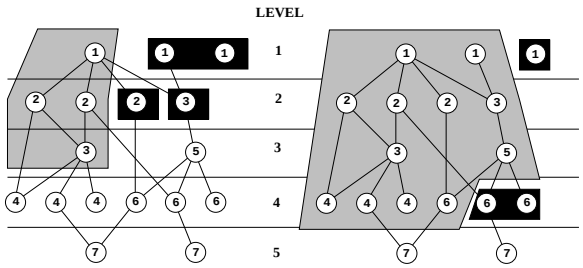

Figure 3: Premature nodes for the DAG in Figure 2 and its p-DFT $T_p = V_1, \ldots, V_7$. On the left, the premature nodes with respect to the prefix V_1, V_2, V_3 are marked in black. On the right, the premature nodes with respect to the prefix $V_1, \ldots, V_6$ are marked in black. The maximum number of premature nodes for this traversal is 4, as required by the prefix V_1, V_2, V_3.

Proof. Let G be a DAG of depth d, and let T_p be a p-traversal based on the given 1-traversal T. We will show that each prefix of T_p has at most $(p-1)\cdot d$ premature nodes; the proof for $(p-1)\cdot(d-1)$ appears in the full paper.

Consider an arbitrary prefix, σ_p, of T_p. Let σ_1 be the longest prefix of T_1 that includes only nodes in σ_p. We say that a level ℓ of G is *fully scheduled* with respect to σ_1 if all nodes in σ_1 at level ℓ are scheduled. We focus our attention on steps of σ_p in which premature nodes were scheduled, showing that in each of these steps, at least one level becomes fully scheduled.

Let $v \in \mathrm{P}(\sigma_p)$ be a premature node, and let i be the (parallel) step of σ_p that schedules v. Let ℓ_i be the smallest level in G that is not fully scheduled with respect to σ_1 at step i of σ_p, and let $\mathrm{US}(\ell_i, \sigma_1, i)$ be the set of ℓ_i-level nodes from σ_1 that are yet unscheduled. Thus, $\ell_i = \min\{\ell : \mathrm{US}(\ell, \sigma_1, i) \neq \emptyset\}$. (Note that levels 1 through $\ell - 1$ may include unscheduled nodes, but these nodes are not in σ_1.) Since σ_1 is a traversal, all nodes in $\mathrm{US}(\ell_i, \sigma_1, i)$ are ready at step i; moreover, their 1-DFT numbers are smaller than that of node v. Hence, all nodes in $\mathrm{US}(\ell_i, \sigma_1, i)$ must be scheduled in step i along with v, and level ℓ_i then becomes fully scheduled with respect to σ_1.

Since for each level ℓ, $\ell = 1, \ldots, d$, there is only one step in σ_p at which level ℓ may become fully scheduled with respect to σ_1, we infer that the number of steps in σ_p in which premature nodes can be scheduled is at most d, as needed. Finally, we note that at most $p-1$ premature nodes can be scheduled at step i, since out of the at most p nodes scheduled at this step, at least one node is from $\mathrm{US}(\ell_i, \sigma_1, i)$ and hence not premature.

The total number of premature nodes with respect to σ_1 that can be scheduled in all the steps of σ_p is therefore at most $d\cdot(p-1)$. Thus each prefix of T_p has at most $(p-1)\cdot d$ premature nodes. ∎

The following lemma shows this upper bound is tight:

Lemma 4.3 (lower bound on premature nodes) *For all $p \geq 1$ and $d \geq 1$, there exists a* DAG*, G, of depth d, and a 1-*DFT*, T, of G such that the maximum number of premature nodes in the greedy p-*DFT *based on T is $(p-1)\cdot(d-1)$.*

Proof. Let G be a forest of depth d with p root nodes such that at each level $i = 1, 2, \ldots, d-1$, only the leftmost node, v_i, at level i has any children, and it has exactly p children. Let v_d be the leftmost child at level d. Consider a 1-DFT, T, that begins by visiting $v_1, v_2, \ldots, v_d$. Let $\sigma_p = V_1, V_2, \ldots, V_{d-1}$ be the first $d-1$ steps (out of d) of the greedy p-DFT of G, *i.e.* for $i = 1, \ldots, d-1$, V_i contains v_i and its $p-1$ siblings. The longest prefix of T that includes only nodes in σ_p is $v_1, \ldots, v_{d-1}$, since v_d is not in σ_p. Thus there are $(p-1)\cdot(d-1)$ nodes in $\mathrm{P}(\sigma_p)$, which is the maximum number of premature nodes in the greedy p-DFT. ∎

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

Lemma 4.4 *For all $p \geq 2$ and $d \geq 4$, there exists a series-parallel* DAG*, G, of depth d, and a 1-*DFT*, T, of G such that the maximum number of premature nodes in the greedy p-*DFT *based on T is $\Omega(p\cdot d)$.*

Proof. (sketch) We define the top half of a graph G; the bottom half is a mirror image of the top half. The top half is a tree such that the root has p children, and for level $i = 2, \ldots, d/2 - 1$, the leftmost child at level i has p children, whereas the $p-1$ other children in the level have a single child (namely, their mirror image in the bottom half). Finally, the p nodes at level $d/2$ each have a single child, their mirror image in the bottom half (at level $d/2+1$). The lemma follows by an argument similar to that in the proof of Lemma 4.3. ∎

4.2 Tight bounds on time and space

We next derive tight bounds on the space requirements.

Theorem 4.5 (upper bound on time and space) *Let G be a dynamically unfolding* DAG *of n nodes and depth d, such that individual tasks allocate at most a constant amount of memory. Let S_1 be the space complexity of a 1-traversal, T, of G. Then for all $p \geq 1$, the space complexity of any p-traversal based on T is bounded by $S_1 + O(p\cdot d)$. As for task bookkeeping space, the space needed to store the frontier of the p-traversal is within $O(p\cdot d)$ of the space needed to store the frontier of the 1-traversal. For greedy p-traversals, the number of steps is at most $n/p + d$.*

Proof. The number of steps follows from Theorem 4.1. As for the program variable space, consider any prefix σ_p of the p-traversal, and the longest prefix σ_1 of T that includes only nodes in σ_p. By Lemma 4.2, there are less than $p\cdot d$ premature nodes, and each can allocate at most a constant amount of space. Since allocations are independent of the traversal, the nodes in σ_1 allocate the same amount of space in both σ_1 and σ_p. Certain deallocations, on the other hand, are credited to the last task in a set of tasks, and hence depend on the traversal. However, any such deallocation occurs in σ_1 only if all tasks in the set have been scheduled, and thus will occur in σ_p. The input space is the same, hence $\mathrm{Space}(\sigma_p) \leq \mathrm{Space}(\sigma_1) + O(p\cdot d) \leq S_1 + O(p\cdot d)$. As for the bookkeeping space, the number of additional frontier nodes at a step is bounded by the number of premature nodes. The theorem follows. ∎

The following result presents a matching lower bound, even when the DAG is restricted to be series-parallel.

Theorem 4.6 (lower bound on time and space) *For all $p \geq 2$ and $d \geq 4$, there exists a series-parallel* DAG*, G, of n nodes and depth d such that the space complexity of the greedy p-*DFT *of G is $S_1 + \Omega(p\cdot d)$, where S_1 is the space for the 1-*DFT*. The individual tasks of G allocate and deallocate at most a constant amount of memory, and this*

*space bound holds even when accounting for task bookkeeping in the 1-*DFT*. For any p-traversal, the number of steps is at least* $\max\{n/p, d\}$*.*

Proof. Consider the DAG defined in the proof of Lemma 4.4, and let the nodes in the top half allocate 1 unit of space, while the nodes in the bottom half deallocate 1 unit of space. Let n be the input space. Then $S_1 \in \Theta(n+d)$. However, for the prefix $\sigma_p = V_1, \ldots, V_{d/2}$ of the greedy p-DFT, $\text{Space}(\sigma_p) = n+1+(d/2-1)\cdot p$. Thus $S_p \in \Omega(n+p\cdot d)$, and hence $S_p \in \Omega(S_1 + p\cdot d)$. The 1-DFT can be implemented in $\Theta(d+p)$ task bookkeeping space, by only allocating for a node just prior to visiting it and deallocating as soon as its child is scheduled. The lower bound on the number of steps is trivial (and well-known). ∎

Handling large allocations. The difficulty with large allocations is that each of the premature nodes at a step may perform a large allocation, thereby exceeding the desired space bound. For example, if in the DAG used to prove Theorem 4.6, the leftmost nodes in the first $d/2$ levels allocate 1 unit of space and the leftmost in the last $d/2$ levels deallocate 1 unit, as before, but the remaining top half nodes allocate k and the remaining bottom half nodes deallocate k, then S_p can be shown to be $S_1 + \Omega(k\cdot p\cdot d)$, for any k. Our approach is to treat each node that allocates k memory as k/m dummy nodes, the last of which performs the actual allocation of size k. This decreases the average penalty for each premature node to be at most m, at the cost of a possible increase in the number of steps, since our traversal may not be greedy. We obtain the following result, whose proof appears in the full paper:

Theorem 4.7 (large allocations) *Let G be a dynamically unfolding* DAG *of n nodes and depth d (with no restriction on the amount of memory an individual task may allocate). Let S_1 be the space complexity of a 1-traversal of G. Then for all $p \geq 1$ and all $m \geq 1$, there is a p-traversal of G with space complexity bounded by $S_1 + O(m\cdot p\cdot d)$ and number of steps at most $n/p + d + S_1/(m\cdot p)$.*

5 Implementation for depth-first p-traversals

The previous section has shown that any greedy p-traversal based on a 1-traversal is efficient in both space and number of steps. Implementing such p-traversals requires some method of determining the relative order in the 1-traversal among the ready nodes at each step, as well as techniques for allocating the scheduled tasks to processors, and for identifying ready nodes. In this section, we present an algorithm for fast implementation of a particular greedy p-traversal, the depth-first p-traversal. Given our results in the previous section, the p-DFT is perhaps the most interesting traversal to consider, since it enables direct comparisons between the space used by a parallel traversal and the space used by the standard sequential traversal.

Our scheduling algorithm applies to dynamically unfolding series-parallel DAGs. Such DAGs arise naturally from languages with nested fork-and-join style parallelism; this includes nearly all the data-parallel languages (both nested and non-nested), as well as many others. A source node in these DAGs corresponds to a task that forks or spawns child tasks; a sink node corresponds to the rejoining of these parallel threads of control. Each source node may spawn an arbitrary number of tasks on each step. This allows for shallower DAGs than if we restricted source nodes to binary fanout; however, it complicates both the scheduling of tasks and their synchronization. Data dependencies are between nodes ordered by paths in the series-parallel DAG. An example mapping of a nested parallelism language to a dynamically unfolding series-parallel DAG is given in the appendix.

Our computation model assumes the outgoing edges from a node are ordered, and we assume here that the standard 1-DFT uses this "left-to-right" order. We also assume that each child of a node has its index among its siblings, corresponding to this left-to-right ordering. Finally, we assume that the number of outgoing edges from a node is revealed when a node is scheduled, even before any space has been allocated for the edges. Our scheduling algorithm performs a constant number of EREW PRAM [JáJ92] steps and a constant number of parallel prefix-sums computations [LF80] for each round of scheduling.

5.1 A stack-based scheduling algorithm

We will use the following property of 1-DFTs on series-parallel DAGs.

Fact 5.1 *Consider a 1-*DFT *of a series-parallel* DAG *G, and let u and v be unordered nodes in G such that u has a lower* DFT *number than v. Then the* DFT *visits any descendant of u that is not a descendant of v prior to visiting v or any descendant of v.*

Proof. The proof follows from the following observation, applied to the lowest common source node of u and v: Let w be a source node in G with $k > 1$ children, $c_1, \ldots, c_k$, in 1-DFT order, and let w' be its associated sink node. For $i = 1, \ldots, k$, let G_i be the subgraph of G consisting of c_i and all nodes that are both descendants of c_i and ancestors of w'. Then the following is a consecutive subsequence of the 1-DFT: node w, then all of G_1, then all of G_2, and so on, until finally all of G_k, followed by node w'. ∎

The Ready Nodes algorithm. We assume for now that we can identify when a node becomes *ready*, *i.e.* when its last parent is scheduled. We defer to Section 5.2 the algorithm and data structures used to implement the identification procedure.

Consider a dynamically unfolding series-parallel DAG G. Let R be an array initially containing the root node of G. Repeat the following two steps until all nodes in G have been scheduled:

1. Schedule the first $\min\{p, |R|\}$ nodes from R, with the ith node in R assigned to processor i.
2. Replace each newly scheduled node by its ready children, in left-to-right order, in place in the array R.

Lemma 5.2 *The Ready Nodes algorithm above produces the greedy p-*DFT *based on the 1-*DFT *of G.*

Proof. We show by induction on the steps of the p-DFT the following invariants: R contains precisely the set of ready nodes, the nodes in R are ordered lowest to highest by their 1-DFT numbers, and the scheduled nodes are a prefix of a greedy p-DFT of G. Initially, the root node is the only ready node, so the invariants hold for the base case. Assume the invariants hold just prior to a step $t \geq 1$. We show that

they hold after step t. Since by the invariants, R contains the ready nodes just prior to step t and the nodes are in 1-DFT order, then the algorithm schedules the ready nodes with lowest DFT numbers. Second, at the end of step t, R contains precisely the ready nodes, since scheduled nodes are removed from R and any newly ready node must have a parent scheduled this step, and hence will be added to R. Third, at the end of step t, the nodes in R are in 1-DFT order. To see this, observe that nodes in R are unordered in G. Hence by Fact 5.1, the left-to-right ordered children that replace a node, u, will have lower DFT numbers than any node, v, to the right of u in R just prior to step t, or any children of v. It follows by induction that the algorithm produces a greedy p-DFT of G. ∎

The problem with the Ready Nodes algorithm is that R may be too large to store without violating our space bounds. Our bounds from Section 4 apply to the number of premature nodes *scheduled*, not the number that are ready. There may be an unbounded number of prematurely ready nodes, since each prematurely scheduled node may have many children. To overcome this difficulty, we can implicitly represent the ready nodes by maintaining the current frontier at each step. Recall that a node is in the frontier if it is scheduled and it has an unscheduled child. However, we would like to be able to extract p ready nodes in $O(p)$ work at each step, and it may require visiting an unbounded number of frontier nodes to identify p ready children. This is due to frontier nodes sharing the same child with non-frontier (*i.e.* unscheduled) nodes, so that the child is not ready, or to an unbounded number of frontier nodes sharing the same ready child.

The P-Ready Nodes algorithm. Our solution is to store, for each ready node, only one of its parent nodes. Define the *last parent* of a node, v, to be the leftmost parent node of v that is scheduled in the step that v becomes ready. Note that if v is a source node, it has only one parent, so distinguishing such a parent node is necessary only for sink nodes. To bound the resource requirements, we use lazy allocation, in which ready nodes are incorporated in the data structure only when they are to be scheduled in the following step. In the remainder of this section, we discuss the algorithm and data structures in more detail.

The P-Ready array. The main component of the data structure is an array, *P-Ready*, which holds the last parent for each ready node. Specifically, there are two types of nodes in *P-Ready*: (i) scheduled source nodes with at least one unscheduled source child; observe the children of such scheduled nodes will be ready; and (ii) scheduled sink nodes that are the last parent of an unscheduled, ready child. For each node v in *P-Ready*, we keep track of the number of its (unscheduled) children $c(v)$. (At each step there may be at most one source node for which only some of its children are scheduled. Handling such a source node is straightforward, and the details appear in the full paper.) As an invariant, we maintain that nodes are represented in the array *P-Ready* in the order of their 1-DFT numbers. The size of array *P-Ready* can be bounded by the space requirement of the p-DFT traversal.

Processor allocation. Compute a prefix-sums computation on the $c(v)$ values for nodes v that are represented in the first p entries of array *P-Ready*. Let the output sequence be in array C. Let i' satisfy $C[i'-1] < p \leq C[i']$ (for simplicity, assume that $p = C[i']$). The children of the first i' nodes are to be scheduled. They are inserted in order into an auxiliary array *Active* of size p: For $i = 1, \ldots, i'$, the representations of the children of node i (in *P-Ready*) are placed in order in entries $C[i-1]+1$ through $C[i]$ of array *Active*. Processor j is allocated to a node represented in *Active*$[j]$, and visits the node; for each node v, the number of its children, $c(v)$, is now revealed. For each sink node, u, in *Active*, if it is not the last parent of its child, set $c(u) = 0$; such items are marked for removal. Use a prefix-sums computation to compact the items in *Active* that are not marked for removal. The first i' entries from array *P-Ready* are cleared, and the contents of *Active* is prepended (in order) to array *P-Ready*.

Lemma 5.3 *The P-Ready Nodes algorithm above produces the greedy p-DFT based on the 1-DFT of G. Its implementation takes a constant number of a p-processor* EREW PRAM *operations, plus a constant number of prefix-sums computations of size p per step of the p-DFT.*

Proof. The proof, which appears in the full paper, uses the fact that the *Active* array is the relevant prefix of the R array used in the Ready Nodes algorithm. ∎

5.2 Task synchronization: Identifying ready nodes

To complete our implementation, we must show how we identify when a node becomes ready. Consider a set of n child tasks that have been spawned in parallel by a parent task (a source node in our DAG with a fanout of n). The *task-synchronization* problem is to quickly detect when the last child completes so that we can restart the computation of the parent (*i.e.* start the sink task corresponding to the source task). Since the computation is dynamic, we do not know ahead of time which child will finish last or how long any of the child computations will take. Furthermore we cannot afford to keep the parent active since this could lead to work-inefficiency (remember that such spawning can be nested).

One way to implement task-synchronization is to associate a counter with the parent which is initialized to n and decremented by each child as it completes. Since multiple children can complete simultaneously, however, this requires a fetch-and-add operation which is expensive both in theory and in practice (especially for an EREW PRAM). A second choice is to build a binary tree when the tasks are spawned which will be used to synchronize as they complete. This, however, requires an $O(\lg n)$ slowdown to go up the tree when synchronizing, and unless dynamic load balancing id used, will also require extra work. In particular this implementation 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. To avoid the problems mentioned above, our implementation is based on the following ideas:
(1) We generate a coordination list among the n children 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 prefix-sums computation to shortcut over all completing neighbors. To make this possible we use properties of the DFT to show that all neighbors that are completing will be adjacent in

our task array. Note that neighbors that are not completing might not be in the task array at all since they might have spawned children and be asleep.
(4) When the last child finishes, it reactivates the parent. If multiple finish simultaneously, then the leftmost reactivates the parent.

Building sink pointers. When a source node v is scheduled (through array *Active*), a representative for its associated sink node, $sink(v)$, is created, and inserted into a set, $\mathcal{S}$. (A source node that is also a sink will be considered for this matter as a pair of source-sink nodes.) The source node will keep a pointer to this representative. When the children of a source node are scheduled, each of them will similarly create a representative for their associated sink nodes. Each child u, upon scheduling, copies the pointer to $sink(v)$ from its parent v, and sets $sink(u)$ to point to $sink(v)$. Note that after the children of v are scheduled, node v is removed from the data structure.

A Coordination linked list. When the children of a source node v are scheduled, we create a linked list among their associated sink nodes, in order. The invariant we maintain is that the list links precisely the nodes from the original list which are still in the data structure; *i.e.*, nodes that are either in the set $\mathcal{S}$ or in the array *P-Ready*. When the list will become empty, the sink node $sink(v)$ will be ready for scheduling. Therefore, when the header of the list (at that particular time) is scheduled, it checks whether the list becomes empty. If it does, then it remains in array *Active*, and hence placed in *P-Ready*. Otherwise, the next unscheduled node in the list will become the header of the list, and the scheduled header is removed from the data structure. Note that each node from the list that is scheduled and is not the list header is promptly removed from the data structure.

Maintaining the Coordination list. The *Coordination* list must be maintained under deletions. The possible difficulty is that a sequence of consecutive nodes in the list can be deleted at the same time. Using pointer jumping for updating the links among the remaining nodes may be rather slow, and an efficient (yet slow) implementation using standard techniques may be quite involved. We present a fast and simple solution to the problem, utilizing the special features of our data structure. The key observation is that if a sequence of two or more adjacent sibling nodes is deleted, then their representatives reside in consecutive entries of array *Active*; hence, updating the *Coordination* list for this deleted sublist is as easy as chaining the first and last entries in the subarray of *Active* containing these representatives. Such local chaining computations can be obtained using a global chaining computation for the nodes represented in array *Active*, and can be done, *e.g.*, by prefix-sums. To indicate why the observation is correct, we note the following: (i) a general property of 1-DFT numbers in series-parallel DAGs is that given two nodes, u and v, that share the same sink child s, and a node w whose 1-DFT number is between that of u and v, then w must be an ancestor of a node v' that shares the same sink child s, such that the 1-DFT number of v' is larger than that of u and smaller or equal to that of v (*i.e.* v' may be v); (ii) nodes are put in their *Coordination* list in the order of their 1-DFT numbers; (iii) nodes in *Active* are ordered by their 1-DFT numbers; (iv) a node only be deleted if it is in array *Active*; and (v) after a node is put in array *Active*, its ancestors cannot be put there. Now, let u and v be adjacent nodes in the *Coordination* list, u prior to v, such that both are deleted. By (iv), both should be in array *Active*. For every node w between u and v in the *Coordination* list, since w was already deleted, then by (iv) and (v), no ancestor of w can be in *Active*. Similarly, since v is in *Active*, none of its ancestors can be in *Active*. The observation follows by (i), (ii) and (iii).

Maintaining the set S. Details left to the full paper.

5.3 Complexity and extensions

Each step of the p-DFT involves at most p unit-time tasks. We say a nested parallelism computation uses *concurrent access primitives* if two or more nodes that are unordered by its DAG read or write the same program variable; if there are no such nodes, the computation uses *exclusive access primitives*. The operations on the data structures in the scheduling algorithm described above for one step of a p-DFT can be implemented by using a constant number of steps on a p-processor EREW PRAM plus a constant number of applications of a prefix-sums computation of size p. We obtain an optimal work implementation on p processors by using a $(p \lg p)$-DFT, thereby amortizing the overhead for the resource allocation at each step (for simplicity, we state the bounds for the case where the sequential space is at worst linear in the sequential running time):

Theorem 5.4 (exclusive access implementation) *Consider a nested parallelism computation with work W, depth D, and sequential space $S_1 \in O(W)$, that uses (only) exclusive access primitives. The above scheduling algorithm can implement the computation in $O(W/p + D \cdot \lg p)$ time and $O(S_1 + D \cdot p \lg p)$ space on a p-processor* EREW PRAM, *or within the same bounds, with high probability, on a p-processor hypercube.*

Proof. (sketch) By Theorem 4.7 (with constant m), there are $O(W/(p \lg p) + D)$ steps in the $(p \lg p)$-DFT we use. Each of these steps can be shown to take $O(\lg p)$ time on p-processors, as follows: A prefix-sums computation of size $p \lg p$ can be implemented on a p-processor EREW PRAM or hypercube in $O(\lg p)$ time [LF80]. Using random hashing techniques, the shared memory of a $(p \lg p)$-processor EREW PRAM can be placed on a p-processor hypercube so that each step of a $(p \lg p)$-processor EREW PRAM can be implemented on the hypercube in $O(\lg p)$ time with high probability [Val90]. Thus the scheduling can be done in $O(\lg p)$ time. Likewise, the $p \lg p$ unit-work tasks can be performed in $O(\lg p)$ time. The program variable space is bounded as needed using Theorem 4.7. The space for the *P-Ready*, *Active*, $\mathcal{S}$, and various temporary arrays used by the scheduling algorithm can also be shown to be within the stated space bounds. The details appear in the full paper. ∎

Faster implementation on the CRCW PRAM. A faster execution can be obtained on the CRCW PRAM, by replacing each prefix-sums computation by either an approximate prefix-sums computations or by a chaining computation. Algorithms for approximate prefix-sums and for chaining are known to take $O(t_{aps})$, where $t_{aps} = \lg \lg p$ in the worst case and $t_{aps} = \lg^* p$ with high probability [BV93, GMV94, GZ95, Rag93]. In order to use the approximate version of the prefix-sums computation, we must allow for a small fraction of null cells in arrays *P-Ready* and *Active*, and allow for a little less than p to be scheduled at each step even if

p are available (as was already allowed to handle large allocations). The details, as well as the proof of the following theorem, appear in the full paper.

Theorem 5.5 (concurrent access implementation) *Consider a nested parallelism computation with work W, depth D, and sequential space $S_1 \in O(W)$, that may use concurrent access primitives. The above scheduling algorithm can implement the computation in $O(W/p+D \cdot t_{aps})$ time and $O(S_1 + D \cdot pt_{aps})$ space on a p-processor* CRCW PRAM*, where t_{aps} is $O(\lg \lg p)$ deterministically or $O(\lg^* p)$ with high probability.*

Memory allocation procedures. The space bounds in the previous theorems 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 the array data structures is straightforward. Memory allocation for the set $\mathcal{S}$ data structure, as well as for program variables, can be done using a dynamic dictionary data structure. An adaptive allocation and deallocation of space, so as to maintain at each step space linear in the number of representatives in the $\mathcal{S}$ data structure, or in the number of program variables, can be implemented with p processors and logarithmic time on the EREW PRAM [PVW83], and in $O(\lg^* p)$ time and linear work with high probability, on a CRCW PRAM [GMV91]. In a recent work [BGM95], we have developed data structures that are weaker than the general dictionary data structure, yet sufficient for our purposes, that obtain a simpler execution on the CRCW PRAM; allowing a logarithmic factor overhead in space, our data structure enables linear work, logarithmic time execution on an EREW PRAM.

These automatic memory allocation procedures can be used in conjunction with the above scheduling algorithm to obtain time-, work-, and space-efficient execution of programs written in languages with fine-grained nested parallelism.

6 Discussion

We derived space and step bounds for executing a general class of online parallel computations. For a more restricted class of nested-parallel computations we described a scheduling algorithm and derived time bounds that include the scheduling cost. For computations with sufficient parallelism ($D \ll S_1$), our space bounds significantly improve previously known bounds.

The work was largely motivated by the desire to generate space-efficient implementations of parallel languages, and in particular the NESL language. The results, however, leave open the question of how practical the techniques are. Here we briefly discuss some issues of practical consideration and describe some variations to our scheduling algorithms that are likely to improve performance. The main two issues we consider are constant factor overheads in the cost of scheduling, and communication costs.

Our schedules assume that each node is a "unit" of work. If these units are just a single operation such as an addition, then the "constant" cost of scheduling the work could significantly outweigh the cost of executing the work. 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. The technique used to break a computation into the appropriate sized blocks is likely to be language dependent and the responsibility of the compiler—at present we can only speculate on how well it can be done. The computational costs of the scheduling itself involves a couple prefix-sums and possibly a few pointer manipulations per task to handle the coordination lists (only threads that are completing need to manipulate pointers), for each round of scheduling (multiple nodes per processor may be scheduled each round). We expect that these costs would be amortized with a block size of a dozen or so instructions.

A second issue is that our scheduling algorithm does not consider machine locality—a particular thread of computation is likely to be scheduled on different processors from one step to the next. A heuristic that could be used to keep threads local to processors is to preferentially schedule tasks from the same thread on the same processor, assuming the tasks remain in the p-DFT. For our stack-based implementation, this would require keeping some dynamic ordering among the processors to properly maintain the stack. How to keep this ordering and how much the heuristic really helps is an area of current research. Another optimization would be to reschedule computation threads only when there is a significant load imbalance among the processors (see, *e.g.* [Sip95]). An open question is whether our algorithm for time-, work-, and space-efficient execution of nested parallel programs can be extended so as to guarantee provably communication-efficient execution as well.

References

[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] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BG95] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, June 1995.

[BGM95] G.E. Blelloch, P.B. Gibbons, and Y. Matias, 1995. In preparation.

[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.

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

[Ble93] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993. Updated version of Technical Report CMU-CS-92-103, Carnegie Mellon U., Jan. 1992.

[Ble95] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 1995. To appear.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal 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.

[BV93] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[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.

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

[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.

[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.

[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.

[HWe91] Paul Hudak, Philip Wadler, and Simon Peyton Jones (eds). *Report on the Functional Programming Language HASKELL*, June 1991. Version 1.1.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[Mat92] Y. Matias. *Highly Parallel Randomized Algorithmics*. PhD thesis, Tel Aviv University, Tel Aviv 69978, 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.

[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.

[Rag93] P.L. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14:371–380, 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.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[Sip95] J. Sipelstein. *Data Representation Optimizations for Collection-Oriented Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1995. To appear.

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

[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.

A Mapping of NESL to dynamic DAGs

NESL is a high-level functional language designed for programming PRAM-like algorithms in the work-time paradigm. The language has a notion of work and depth that can be derived from the costs of the primitives and composed across expressions. In this appendix, we describe the core of the NESL language, explain the complexity model, and relate it to our DAG model. We show (informally) that the computation DAGs of NESL programs are series-parallel, and that the number of nodes, W, and the depth, D, of the DAGs are within a constant factor of the work and depth derived from the NESL cost model.

NESL language is a functional language with strong polymorphic typing and call-by-value semantics (it is loosely based on ML).[1] The syntax of the core language is given in Figure 4.

The NESL language supports parallelism by means of operations on sequences—one dimensional arrays. The main parallel construct is a parallel *map*, which permits any function to be mapped over the elements of a sequence. The map construct uses a set-like syntax similar to set-formers in SETL [SDDS86] or list-comprehensions in Haskell [HWe91]. In particular, the notation $\{e_1 : x \texttt{ IN } A\}$ can be read as "for

[1] The full Nesl language has some nonfunctional features, such as I/O.

```
funcdef  ::=  FUNCTION ident vars = exp

vars     ::=  ident | vars,vars | ( vars )

exp      ::=  LET vars = exp IN exp
           |  IF exp THEN exp ELSE exp
           |  {exp :  vars IN exp}
           |  exp exp | exp binop exp
           |  const | ident | ( exp )
```

Figure 4: Syntax of the core of the NESL language. Above, **ident** is any variable identifier, **const** is any constant in the language, and **binop** is one of the built in binary operators, which include standard arithmetic functions such as `+` and `*`, as well as the , operator for creating pairs. The expression *exp exp* signifies function application—the first expression must evaluate to a function and typically is just the function name.

x bound to each element of the sequence A execute the expression e_1, and return a sequence of results". In addition to the parallel map construct, NESL supplies the built-in functions on sequences listed in Figure 5. These are a minimal set of primitives in the sense that none can be removed without effecting the asymptotic complexity of programs.

Operation	Work	Depth
`#(a)`	1	1
`index(n)`	n	1
`elt(a,i)`	1	1
`write(d,a)`	(#a + #d) or #a	1
`flatten(a)`	#result	1
`partition(a,l)`	#a + #l	1

Figure 5: Built in NESL function and their work and depth. The `#` function takes the length of a sequence; `index` returns an index vector of length `n`; `elt` removes an element from a sequence; `write` writes a set of value into a sequence; `flatten` flattens a nested sequence; and `partition` partitions a sequence.

The mapping of NESL onto our dynamically unfolding DAGs can be defined recursively by the structure of a program as it executes. In particular, the execution of every expression in NESL corresponds to a series-parallel DAG with a single source and sink. We can then look at the various syntactic forms and specify how they compose the DAGs of their constituent parts. The composition rules are expressed in Figure 6. It should be noted that for the conditional `IF` statement, the DAG will depend on the runtime execution. We only show the DAG for one of the built-in functions (`index`) but the others are similar.

The NESL language specifies the work and depth costs of all of its primitives and supplies a set of rules for composing work and depth cost measures across expressions. The work and depth of executing any program is therefore well defined. The *Work* complexity for a NESL program is defined recursively, as follows. The work complexity of a predefined scalar function is 1, the work complexity of a predefined sequence function are specified in Figure 5, and the work complexity for all other expressions can be composed as specified in Figure 7. A similar set of rules can be used for composing depth. The only difference is in the *map* rule. The relationship between the work and depth calculated by the rules and the size and depth of the DAGs can be derived in a straightforward manner based on the rules in Figure 7 and the mapping in Figure 6. In particular they can be bounded both ways by constant factors.

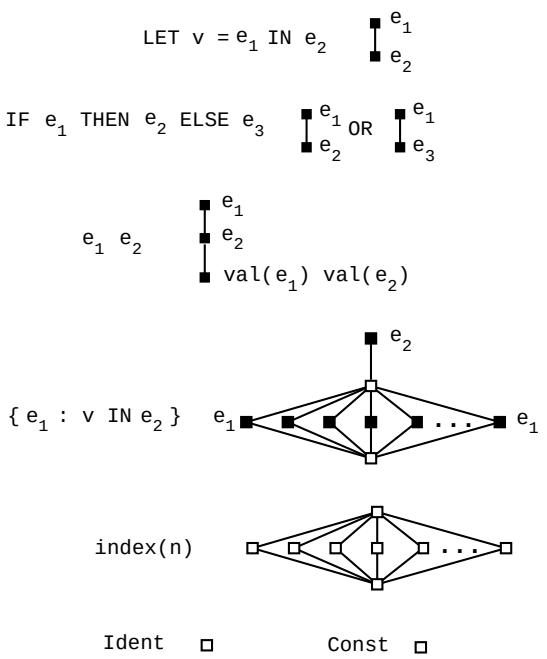

Figure 6: Mapping from NESL to DAGs. The dark nodes represent subexpressions, the light nodes represent units of work.

$W(\texttt{LET } v = e_1 \texttt{ IN } e_2)$	$= 1 + W(e_1) + W(e_2)$
$W(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3)$	$= 1 + W(e_1) + \begin{cases} W(e_2) & e_1 \text{ is true} \\ W(e_3) & \text{otherwise} \end{cases}$
$W(\{e_1 : v \texttt{ IN } e_2\})$	$= 1 + W(e_2) + \sum_{a \texttt{ in } e_2} W(e_1(a))$
$W(e_1\ e_2)$	$= 1 + W(e_1) + W(e_2) + W(val(e_1)val(e_2))$
$W(\textbf{ident})$	$= 1$

Figure 7: The *work* composition rules for each type of expression in NESL. The rules for *depth* are identical except for the map rule, for which maximum is used instead of summation. These equations are for the CRCW variant of the cost models. The EREW variant has an additional term in the map rule to account for copying constants across the elements [Ble93].