Space-Efficient Scheduling of Parallelism with Synchronization Variables

Guy E. Blelloch
Carnegie Mellon
guyb@cs.cmu.edu

Phillip B. Gibbons
Bell Laboratories
gibbons@research.bell-labs.com

Yossi Matias
Bell Laboratories
matias@research.bell-labs.com

Girija J. Narlikar
Carnegie Mellon
girija@cs.cmu.edu

Abstract

Recent work on scheduling algorithms has resulted in provable bounds on the space taken by parallel computations in relation to the space taken by sequential computations. The results for online versions of these algorithms, however, have been limited to computations in which threads can only synchronize with ancestor or sibling threads. Such computations do not include languages with futures or user-specified synchronization constraints. Here we extend the results to languages with synchronization variables. Such languages include languages with futures, such as Multilisp and Cool, as well as other languages such as ID.

The main result is an online scheduling algorithm which, given a computation with w work (total operations), σ synchronizations, d depth (critical path) and s_1 sequential space, will run in $O(w/p + \sigma\log(pd)/p + d\log(pd))$ time and $s_1 + O(pd\log(pd))$ space, on a p-processor CRCW PRAM with a fetch-and-add primitive. This includes all time and space costs for both the computation and the scheduler. The scheduler is non-preemptive in the sense that it will only move a thread if the thread suspends on a synchronization, forks a new thread, or exceeds a threshold when allocating space. For the special case where the computation is a planar graph with left-to-right synchronization edges, the scheduling algorithm can be implemented in $O(w/p+d\log p)$ time and $s_1 + O(pd\log p)$ space. These are the first non-trivial space bounds described for such languages.

1 Introduction

Many parallel languages allow for dynamic fine-grained parallelism and leave the task of mapping the parallelism onto processors to the implementation. Such languages include both data-parallel languages such as HPF [24] and NESL [3], and control-parallel languages such as Multilisp [23], ID [1], SISAL [18] and Proteus [28]. Since there is often significantly more parallelism expressed in these languages than there are processors, the implementation must not only decide onto which processors to schedule computations, but in what order to schedule them. Furthermore, since the parallelism is often dynamic and data-dependent, these decisions must be made online while the computation is in progress. The order of a schedule can have important consequences on both the running time and the space used by an application.

There has been a large body of work on how to schedule computations so as to minimize running time. This work dates back at least to the results by Graham [21]. More recently, in addition to time, there has been significant concern about space. This concern has been motivated in part by the high memory usage of many implementations of languages with dynamic parallelism, and by the fact that parallel computations are often memory limited. A poor schedule can require exponentially more space than a good schedule [11]. Early solutions to the space problem considered various heuristics to reduce the number of active threads [10, 23, 34, 16, 26].

More recent work has considered provable bounds on space usage. The idea is to relate the space required by the parallel execution to the space s_1 required by the sequential execution. Burton [11] first showed that for a certain class of computations the space required by a parallel implementation on p processors can be bound by $p \cdot s_1$ (s_1 space per processor). Blumofe and Leiserson [8, 9] then showed that this space bound can be maintained while also achieving good time bounds. They showed that a fully strict computation that executes a total of w operations (work) and has a critical path length (depth) of d can be implemented to run in $O(w/p + d)$ time, which is within a constant factor of optimal. These results were used to bound the time and space used by the Cilk programming language [7]. Blelloch, Gibbons and Matias [4] showed that for nested computations, the time bounds can be maintained while bounding the space by $s_1 + O(pd)$, which for sufficient parallelism is just an additive factor over the sequential space. This was used to bound the space of the NESL programming language [5]. Narlikar and Blelloch [30] showed that this same bound can be achieved in a non-preemptive manner (threads are only moved from a processor when synchronizing, forking or allocating memory) and gave experimental results showing the effectiveness of the technique. All this work, however, has been limited to computations in which threads can only synchronize with their sibling or ancestor threads. Although this is a reasonably general class, it does not include languages based on futures [23, 27, 12, 14, 13], languages based on lenient or speculative evaluation [1, 32], or languages with general user-specified synchronization constraints [33].

In this paper we show how to extend the results to support synchronization based on write-once synchronization variables. A *write-once synchronization variable* is a variable (memory location) that can be written by one thread and read by any number of other threads. If it is read before it is written, then the reading thread is suspended until the variable is written. Pointers to such synchronization variables can be passed around among threads and synchronization can take place between two threads that have pointers to the variable. Such synchronization variables can be used to implement futures in such languages as Multilisp [23], Mul-T [27], Cool [14] and OLDEN [13]; I-structures in ID [1]; events in PCF [38]; streams in SISAL [18]; and are likely to be helpful in implementing the user-specified synchronization constraints in Jade [33]. We model computations that use synchronization variables as directed acyclic graphs (DAGs) in which each node is a unit time action and each edge represents either a control or data dependence between actions. The work of a computation is then measured as the number of nodes in the DAG and the depth as the longest path.

The main result of this paper is a scheduling algorithm which, given a parallel program with synchronization variables such that the computation has w work, σ synchronizations, d depth and s_1 sequential space, executes the computation in $s_1+O(pd\log(pd))$ space and $O(w/p+\sigma\log(pd)/p+d\log(pd))$ time on a p-processor CRCW PRAM with a fetch-and-add primitive [20]. This includes all time and space costs for both the computation and the scheduler. This algorithm is work-efficient for computations in which there are $\Omega(\log(pd))$ units of work per synchronization (on average). In addition, we show that if the DAG is planar, or close to it, then the algorithm executes the computation in $s_1+O(pd\log p)$ space and $O(w/p+d\log p)$ time, independent of the number of synchronizations. Planar DAGs are a more general class of DAGs than the computation DAGs considered in [8, 9, 4]. Previously, no space bounds were known for computations with synchronization variables, even in the case where the DAGs are planar.

As with previous work [4, 29], the idea behind the implementation is to schedule the threads in an order that is as close as possible to the sequential order (while still obtaining good load balance across the processors). This allows us to limit the number of threads that are executed prematurely relative to the sequential order, and thus limit the space. An important contribution of this paper is an efficient technique for maintaining the threads prioritized by their sequential execution order in the presence of synchronization variables. This is more involved than for either the nested parallel or fully strict computations considered previously, because synchronization edges are no longer "localized" (a thread may synchronize with another thread that is not its sibling or ancestor — see Figure 1). To maintain the priorities we introduce a black-white priority-queue data structure, in which each element (thread) is colored either black (ready) or white (suspended), and describe an efficient implementation of the data structure based on 2-3 trees. In addition, we use previous ideas [22] for efficiently maintaining the queues of suspended threads which need to be associated with each synchronization variable. As with [29], our scheduler is asynchronous (its execution overlaps asynchronously with the computation) and non-preemptive (threads execute uninterrupted until they suspend, fork, allocate memory or terminate).

For planar DAGs, we prove that a writer to a synchronization variable will only wake up a reader of the variable that has the same priority as itself, relative to the set of ready threads. This enables a more efficient implementation for planar DAGs. Planar DAGs appear, for example, in producer-consumer computations in which one thread produces a set of values which another thread consumes.

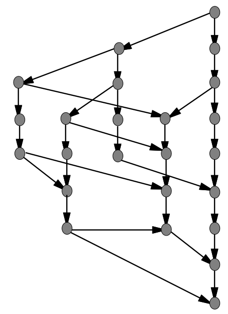

Figure 1: An example (non-planar) DAG for a computation with synchronization variables. Threads are shown as a vertical sequence of actions (nodes); each right-to-left edge represents a fork of a new thread, while each left-to-right edge represents a synchronization between two threads.

2 Programming model

As with the work of Blumofe and Leiserson [8], we model a computation as a set of threads, each comprised of a sequence of instructions. Threads can fork new threads and can synchronize through the use of write-once synchronization variables (henceforth just called synchronization variables). All threads share a single address space. We assume each thread executes a standard RAM instruction set augmented with the following instructions. The fork instruction creates a new thread, and the current thread continues. The allocate(n) instructions allocates n consecutive memory locations and returns a pointer to the start of the block. The sv-allocate instruction allocates a synchronization variable and returns a pointer to it. The free instruction frees the space allocated by one of the allocate instructions (given the pointer to it). The standard read and write instructions can be used to read from and write to a synchronization variable as well as regular locations. Each synchronization variable, however, can only be written to once. A thread that performs a read on an unwritten synchronization variable suspends itself; it is awakened when another thread performs a write on that variable. We assume there is an end instruction to end execution of a thread.

In this model a future can be implemented by allocating a synchronization variable, forking a thread to evaluate the future value, and having the forked thread write its result to the synchronization variable. I-structures in ID [1] can similarly be implemented with an array of pointers to synchronization variables and a fork for evaluating each value. Streams in SISAL [18] can be implemented by associating a synchronization variable with each element (or block of elements) of the stream.

We associate a directed acyclic graph (DAG) called a *computation graph* with every computation in the model. The computation graphs are generated dynamically as the computation proceeds and can be thought of as a trace of the computation. Each node in the computation graph represents the execution of a single instruction, called an *action*[1], and the edges represent dependences among the actions. There are three kinds of dependence edges in a computation graph: thread edges, fork edges and data edges. A thread is modeled as a sequence of its actions connected by *thread edges*. When an action a_1 within a thread τ_1 forks a

[1] We assume that every action requires a single time step to be executed.

new thread τ_2, a *fork edge* is placed from a_1 to the first action in τ_2. When an action a_1 reads from a synchronization variable, a *data edge* is placed from the action a_2 that writes to that variable to a_1. For example, in Figure 1, the vertical edges are thread edges, the right-to-left edges are fork edges, and the left-to-right edges are data edges. The time costs of a computation are then measured in terms of the number of nodes in the computation graph, called the *work*, the sum of the number of fork and data edges, called the number of *synchronizations*, and the longest path length in the graph, called the *depth*.

We say that a thread is *live* from when it is created by a fork until when it executes its end instruction, and a live thread is *suspended* if it is waiting on a synchronization variable. We also assume that computations are deterministic, that is, the structure of the computation graph is independent of the implementation.

Serial space. To define the notion of serial space s_1 we need to define it relative to a particular serial schedule. We define this schedule by introducing a *serial priority order* on the live threads—a total order where $\tau_a \succ \tau_b$ means that τ_a has a higher priority than τ_b. To derive this ordering we say that whenever a thread τ_1 forks a thread τ_2, the forked thread will have a higher priority than the forking thread ($\tau_2 \succ \tau_1$) but the same priority relative to all other live threads. The serial schedule we consider is the schedule based on always executing the next action of the highest-priority non-suspended thread. This order will execute a depth-first topological sort of the computation graph. We call such a serial schedule a 1DF-schedule. Serial implementations of the languages we are concerned with, such as PCF, ID, and languages with futures, execute such a schedule. For example, with futures, the 1DF-schedule corresponds to always fully evaluating the future thread before returning to the body. Figure 2 shows an example computation graph, in which each action (node) is labeled in the order in which it is executed in a 1DF-schedule. In the rest of the paper, we refer to this depth-first order as the serial execution order.

Let $(a_1, a_2, \ldots, a_T)$ be the actions of a T node computation graph, in the order in which they are executed in a 1DF-schedule. To define the serial space s_1, we associate a weight $w(a)$ with each action (node) a of the computation graph. For every action a_i that corresponds to a sv-allocate we set $w(a_i) = 1$ on the node representing a_i; for every action a_i that corresponds to an allocate(n) instruction we set $w(a_i) = n$; for every free we place the negative of the amount of space that is deallocated by the free; and for every other action we set the weight to zero. Then the serial space requirement s_1 for an input of size N is defined as $s_1 = N + \max_{i=1,\ldots,T}\left(\sum_{j=1}^{i} w(a_j)\right)$.

3 Outline of the Paper

In the remainder of the paper we are concerned with the efficient implementation of the programmer's model, discussed in the previous section. We first introduce the notion of a task graph in Section 4. A task is a sequence of actions within a single thread that gets executed without preemption. The main purpose of the task graph is to model non-preemption in the scheduler so that we can formally specify under what conditions the scheduler will preempt threads. The task graph also serves to help prove our time and space bounds. Section 5 then describes the scheduler, including how it breaks threads into tasks and how it maintains the threads in the appropriate priority order using a balanced tree. Section 6 proves the space and time bounds for the resulting schedules. Section 7 describes a more efficient algorithm that works for planar or near-planar graphs.

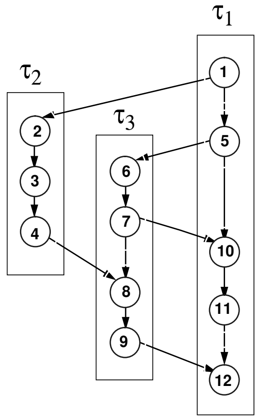

Figure 2: An example computation graph in which actions are labeled with their serial execution order. Left-to-right edges represent the synchronizations between threads, while right-to-left edges are the fork edges. Threads are shown as solid rectangles around the nodes, while consecutive nodes of a thread are grouped into tasks (defined in Section 4), which are shown as dashed ovals. Thread τ_1 forks thread τ_2 and then forks thread τ_3; therefore, $\tau_2 \succ \tau_3 \succ \tau_1$.

4 Task Graphs

Schedules of computation graphs represent executions in which the unit of scheduling is a single action. Therefore, an algorithm that generates such schedules may map consecutive actions of a thread onto different processors. This can result in high scheduling overheads and poor locality. To overcome these problems, we increase the granularity of the scheduled unit by grouping sequences of actions of a thread into larger *tasks*; each task is executed non-preemptively on one processor. We call the DAG of tasks a *task graph*. Task graphs are created dynamically by the implementation described in this paper; the programmer need not be aware of how the actions are grouped into tasks. The example in Figure 2 shows the grouping of actions into tasks.

Definitions. A *task graph* $G_T = (V, E)$ is a DAG with weights on the nodes and edges. Each node of G_T is a *task*, and represents a series of consecutive actions that can be executed without stopping. Each task $v \in V$ is labeled with a nonnegative integer weight $t(v)$, which is the *duration* of task v (the time required to execute v, or the number of actions represented by v). For every edge $(u, v) \in E$, we call u the *parent* of v, and v the *child* of u. Each edge $(u, v) \in E$ has a weight $l(u, v)$, which represents the minimum latency between the completion of task u and the start of task v. (In this paper, we will consider latencies due to the scheduling process.)

Once a task v is scheduled on a processor, it executes to completion in $t(v)$ timesteps. The *work* w of a task graph G_T is defined as $w = \sum_{v \in V} t(v)$. The *length* of a path in G_T is the sum of the durations of the tasks along the path. Similarly, the *latency-weighted length* is the sum of the durations of the tasks plus the sum of the latencies on the edges along the path. The *depth* d of G_T is the

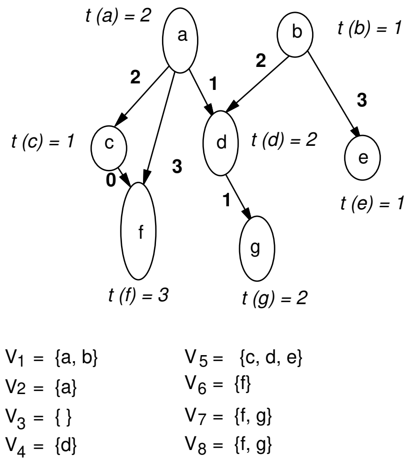

Figure 3: An example task graph and a p-schedule for it (with $p \geq 3$). Each oval represents a variable-length task. For each task v, $t(v)$ denotes the duration of the task, that is, the number of consecutive timesteps for which v must execute. Each edge is labeled (in bold) with its latency. For $i = 1, \ldots, 8$, V_i is the set of tasks being executed during timestep i of the p-schedule.

maximum over the lengths of all the paths in G_T, and the *latency-weighted depth* d_l is the maximum over the latency-weighted lengths of all the paths in G_T.

As with computation graphs, the parallel execution of a computation on p processors can be represented by a *p-schedule* $\mathcal{S}_p$ of its task graph. A p-schedule of a task graph $G_T = (V, E)$ is a sequence $(V_1, V_2, \ldots, V_T)$ which satisfies the following conditions:

1. $V = \bigcup_{i=\{1,\ldots,T\}} V_i$
2. We say v is *scheduled* in timestep i if $v \notin V_{i-1}$ and $v \in V_i$. If v is scheduled at timestep i, then $v \in V_j \iff i \leq j < i + t(v)$.
3. $\forall i \in 1, \ldots, T$, $|V_i| \leq p$.
4. A task v is *completed* in timestep i if $v \in V_i$ and $v \notin V_{i+1}$. $\forall (u, v) \in E$, if u is completed in timestep i, and v is scheduled in timestep j, then $j > i + l(u, v)$.

A task v is *ready* when all its parent tasks have been completed and the latencies on all edges into v have been satisfied, but v is yet to be scheduled. We say that a p-schedule is *greedy* [8] if $\forall i \in 1, \ldots, T$, $|V_i| < p$ implies all ready tasks are scheduled on that step. Figure 3 shows an example task graph and a greedy p-schedule for it. The proof of the following theorem can be found in [29].

Theorem 4.1 *Given a task graph G_T with w work and latency-weighted depth d_l, any greedy p-schedule of G_T will require at most $w/p + d_l$ timesteps.* ∎

This is within a factor of two of the trivial lower bound of $\max(w/p, d_l)$.

The scheduling algorithm must decide which of the ready tasks to schedule in each step of a schedule. If tasks are assigned priorities, and at every step, the tasks scheduled are the ready tasks with the highest priorities, we call the resulting schedule a *prioritized schedule*. This definition is similar to the class of list schedules described in [17]. Let $\mathcal{S}_1$ be any 1-schedule for a task graph with n tasks, and let $v_1, v_2, \ldots, v_n$ be the tasks in the order they appear in $\mathcal{S}_1$. As defined in [4], we say a prioritized p-schedule is *based* on $\mathcal{S}_1$ if the relative priorities of tasks are based on their serial execution order: $\forall i, j \in \{1, \ldots, n\}$, $i < j \Rightarrow priority(v_i) > priority(v_j)$.

Modeling space with task graphs. As with computation graphs, we associate weights with each task to model space allocations in a task graph. However, since each task may contain multiple allocations and deallocations, we introduce two integer weights for each task v in a task graph G_T: the *net memory allocation*, $n(v)$, and the *memory requirement*, $h(v)$. The weight $n(v)$ is the difference between the total memory allocated and the total memory deallocated in v, and may be negative if the deallocations exceed the allocations. The weight $h(v)$ is the non-negative high-water mark of memory allocation in v, that is, the maximum memory allocated throughout the execution of task v. The task v can be executed on a processor given a pool of at least $h(v)$ units of memory. If C_i is the set of tasks that have been completed at or before timestep i of $\mathcal{S}$, that is, $C_i = \{v \in V_j \mid (j \leq i) \text{ and } (v \notin V_{i+1})\}$, then the space requirement of $\mathcal{S}$ for an input of size N is defined as

$$space(\mathcal{S}) \leq N + \max_{i=1,\ldots,T} \left(\sum_{v \in C_i} n(v) + \sum_{v \in V_i - C_i} h(v) \right) \quad (1)$$

Note that this is an upper bound on the space required by the actual execution, since there may be tasks v executing during a timestep that are not presently at their respective high-water marks $h(v)$ of memory allocation. However, if $\mathcal{S}$ is a serial schedule, then the above expression for $space(\mathcal{S})$ is an exact value for the space requirement of the execution it represents.

5 The scheduling algorithm

In this section, we first describe how multithreaded computations are broken into task graphs. Next, we present an online and asynchronous scheduling algorithm that generates such task graphs and space-efficient schedules for them, and finally describe an efficient implementation of the scheduler. For now we assume that every action in a parallel computation with synchronization variables allocates at most a constant K ($K \geq 1$) units of space. In Section 6 we will first prove the space and time bounds of the generated schedule for such computations, and then show how to relax this assumption without affecting our space bounds.

Creating task graphs for computations. Let G be the computation graph representing the computation and G_T be the task graph representing the same computation. As with G, the task graph G_T is created dynamically as the computation proceeds and contains data, fork, and thread edges. A thread is broken into a new task when it performs one of the following actions: (1) a read on a synchronization variable when the variable is not ready (i.e., there has been no previous write to it), (2) a write on a synchronization variable with waiting threads (i.e., there has been at least one previous read on the variable), (3) a fork, and (4) an allocation that causes the memory requirement of the current task to exceed some threshold M, defined later. These interruptions in a thread are denoted *breakpoints*. A fork edge between two actions in G becomes a fork edge between the corresponding tasks in G_T. G_T has a data edge from a task, u, with a write to a synchronization variable to every task, v, with a corresponding read such that the read takes place before the write. The latencies on the edges of G_T are determined by the time overhead incurred by the scheduling algorithm for maintaining ready tasks in the system (their values are explained in Section 6).

5.1 Algorithm Async-Q

The job of the scheduling algorithm is to efficiently schedule the tasks generated as described above onto the processors. The algorithm is online in the sense that it has to run while the computation is proceeding since neither the task graph nor the serial schedule are known. The algorithm we present uses a set of *worker* processors to run the tasks and a separate set of processors to execute the *scheduler*. In the conclusion we mention how the processors might be shared. The worker processors run asynchronously with each other and with the scheduler. They only synchronize with the scheduler through two FIFO queues, called Q_{in} and Q_{out}, and only when reaching a breakpoint. The processors executing the scheduler run synchronously with each other and are responsible for maintaining the set of all live threads L prioritized by their serial execution order, and for communicating with the workers through the FIFO queues. During a computation each live thread can either be *active* (currently in Q_{in}, Q_{out} or being executed by a worker processor), *ready* if it is ready to execute but not active, or *suspended* if it is waiting on a synchronization variable. Let $R \subseteq L$ be the set of ready threads in L.

Figure 4 specifies the algorithm Async-Q for both the workers and the scheduler, and Figure 5 shows the migration of threads between Q_{in}, Q_{out}, the worker processors and the set of live threads maintained by the scheduler. It is important to realize that the scheduler is executed by the set of scheduler processors while the worker is executed by each individual worker processor. We call each iteration of the scheduler loop a *scheduling iteration*. In the last timestep of each scheduling iteration, the scheduler inserts tasks into Q_{out}, which are available to the workers at the next timestep. Initially L contains only the root thread, which is in the ready state.

To implement the scheduling algorithm on p processors we assign a constant fraction αp of the processors $(0 < \alpha < 1)$ to the scheduler computation, and the remaining $(1-\alpha)p$ processors as workers.

begin *worker*
while *(there exist threads in the system)*
  τ := *remove-thread*(Q_{out});
  execute τ *until it reaches a breakpoint or terminates;*
  insert-thread(τ, Q_{in});
end *worker*

begin *scheduler*
while *(there exist threads in the system)*
  T := *remove-all-threads*(Q_{in});
  for *each thread* τ *in* T
    if τ *has written to a synchronization variable* ϑ,
      mark all threads suspended on ϑ *as ready in* L;
    else if τ *has terminated, delete* τ *from* L;
    else if τ *has* forked, *add the new thread to* L;
    else if τ *has suspended on a synchronization variable,*
      mark as suspended in L;
    else *retain* τ *as ready in* L;
  mark $\min(|R|, q_{max} - |Q_{out}|)$ *ready threads with highest*
    priorities in L *as active and insert them into* Q_{out};
end *scheduler*

Figure 4: Algorithm Async-Q. The worker and scheduler computations execute asynchronously in parallel. q_{max} is the maximum size of Q_{out}, and $|Q_{out}|$ is the number of tasks in Q_{out}. The function *remove-thread*(Q_{out}) busy-waits until a thread becomes available and is removed from Q_{out}.

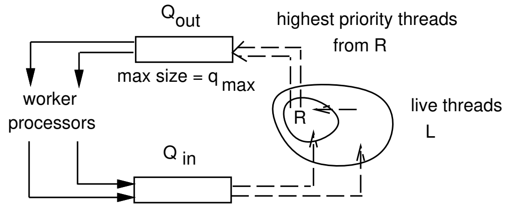

Figure 5: The movement of threads between the processors and the scheduling queues. The dashed lines indicate the movement of threads by the scheduler, whereas the solid lines indicate the movement of threads by the worker processors (the processors that execute the threads).

5.2 Implementation of the scheduler

The main job of the scheduler is to maintain the threads in L prioritized by their serial execution order, so that it can efficiently pick the ready threads with the highest priorities in every scheduling iteration. For computations with synchronization variables this is more involved than in previous work [29] which handled computations with nested parallelism. With nested parallelism the threads L could be kept in priority order using a simple array in which the scheduler only needed to access one end of the array. This is not in general possible with synchronization variables since a computation can have many suspended threads with higher priorities than a ready thread, requiring the scheduler to search through a large number of threads to find the set of highest-priority ready threads. In this section we present an implementation of L as a *black-white priority queue* which allows efficient access. In the queue each element is either black (representing ready threads) or white (representing active or suspended threads). Since the absolute serial execution order of the threads is not known for online computations, the queue maintains threads according to their relative ordering. Figure 6 lists the operations supported by the priority queue.

We implement the black-white priority queue using a 2-3 tree in which the elements (threads) are kept at the leaves and are ordered by their relative priorities from left to right (the leftmost leaf has the highest priority)[2]. As with standard 2-3 trees, all leaves are kept at the same level and all internal nodes have 2 or 3 children. Instead of keeping keys at each internal node, however, we keep a count. The count represents the number of black elements in the subtree rooted at the node. Keys are not used since we will not need to search the tree—in fact, since the absolute serial ordering is not known, there are no key values to use. For this data structure, we can prove the following bound on the time required to execute operations on its elements.

Lemma 5.1 *Using a black-white priority queue of size n, each of the four operations from Figure 6 on m elements can*

[2] Other balanced tree implementations that support fast parallel inserts, deletes and lookups might be used instead of 2-3 trees

Queue operation	Function in scheduler
Recolor white element to black	Awaken suspended thread or convert thread from active to ready
Split element into two adjacent elements	Fork new threads
Delete element	Remove terminated thread
Select m black elements with highest priority and recolor to white	Move highest-priority ready threads to Q_{out}

Figure 6: Operations supported by the priority queue and their corresponding functions in the scheduler.

be implemented to run in $O(\frac{m}{p}\log n)$ time on an p-processor EREW PRAM.

Proof: For all the operations, we assume that we have direct pointers to the m elements in the 2-3 tree. Consider each of the four operations. (1) For recoloring m white elements to black we start with the m elements at the leaves and work our way up the tree incrementing the counters appropriately. This is executed in parallel level by level. When paths from two (or three) recolored elements to the root meet at an internal node, the two (three) increments are combined, and only one proceeds to the next level. (2) For splitting m elements we can use a procedure similar to Paul, Vishkin and Wagener's [31] procedure for inserting into 2-3 trees. Since each leaf can only split into two leaves, the splitting may convert a node just above the leaf level into a 4, 5 or 6 node which then needs to be split. This in turn may split nodes at the next level, and the splitting can go up the tree level by level to the root. The counts are updated along the way as before. (3) For deleting elements we can again use a procedure similar to the one used by Paul, Vishkin and Wagener. Since a set of contiguous elements may need to be deleted, the pipelined version of their algorithm is used. (4) For selecting the m highest-priority black elements we can start at the root and proceed downwards, using the counts to locate the leftmost m elements, and searching the required subtrees in parallel. Operations (1) and (2) can be implemented in $O(\frac{m}{p}\log n)$ time on an EREW PRAM by simply having each of the m elements walk its way up the tree in parallel. Operation (3) will run within the same bounds since the pipeline will only create a delay of $O(\frac{m}{p}\log n)$. Operation (4) can be implemented on a EREW PRAM by starting with one job at the root and then forking one, two or three jobs at each node depending on which nodes need to be searched to find the desired black elements. Since we know how many black elements are within each subtree we can assign a proportional number of processors to search that subtree and only one needs to look at the root. Since the depth of the tree is bound by $\log n$ and the total number of forked jobs at completion is bound by m the time is bound by $O(\frac{m}{p}\log n)$. ∎

The other operation the scheduler needs to perform is to handle the queues of threads waiting on synchronization variables. We use an array for each queue, which we will call a *synchronization queue* [22].

Aiming for an efficient implementation of the scheduler, we set q_{max} to be p and the maximum task space M to be $\log(pd)$. We can show that no more than $P = (2+1-\alpha)p < 3p$ threads can be active at any time, and all the P active threads can end up in Q_{in} before the start of a scheduling iteration. Therefore, a step of the scheduler will need to split, delete, and select at most $3p$ elements. However it could recolor a larger number since awakening threads can potentially recolor all the white nodes to black.

Lemma 5.2 *The sum of the time taken by any m scheduling iterations is bound by $O((\sigma/p + m)\log|L|)$ time on a αp processor* CRCW PRAM *with fetch-and-add.*

Proof: The time taken by a single scheduling iteration that awakens n suspended threads is $O((n/p+1)\log|L|)$ not including the time for suspending threads (adding them to synchronization queues). This is based on Lemma 5.1 and previous bounds that state that n threads can be reawakened from a set of synchronization queues [22] in $O(n/p)$ time on a CRCW PRAM with fetch-and-add. Since at most σ threads will be awakened during a computation, the sum of the above time over m scheduling iterations is bound by $O((\sigma/p + m)\log|L|)$. Again based on previous results [22] the cost of suspending threads (placing them in the synchronization queues) can be amortized against the cost of reawakening them. Since we are including the cost of reawakening in the above bounds, the total cost of m scheduling iterations including the suspension cost is bound by $O((\sigma/p + m)\log|L|)$. ∎

6 Space and time bounds

In this section, we first define a class of space-efficient schedules called Q-prioritized schedules, by presenting a high-level description of scheduling algorithms that generate them. Such schedules are based on serial schedules but are not completely prioritized; this deviation from priorities allows the schedules to be generated in a simple and efficient manner. We then prove the space and time bounds for schedules generated by algorithm Async-Q by showing that they are Q-prioritized schedules.

Q-prioritized schedules. Q-prioritized schedules are the class of schedules that are generated by any scheduling algorithm that conforms to the following rules.

1. The scheduling algorithm maintains a FIFO work queue that contains a subset of the ready tasks. Let q_{max} be the maximum size of the work queue, and let Q_t be the tasks in it at timestep t. Idle processors take tasks from the head of the work queue and execute them.

2. Let R_t be the set of all ready tasks at timestep t of the execution, and let $R'_t = R_t - Q_t$ be the subset of R_t not in the work queue. Let T_t be the set of tasks moved from R'_t to the work queue at timestep t. Then, $|T_t| \leq \min(q_{max} - |Q_t|, |R'_t|)$, and T_t is the subset of R'_t with the highest priorities. No tasks are moved back from Q_t to R'_t.

As with prioritized schedules, priorities are based on the serial execution order of the tasks. Tasks inserted into the work queue at timestep t are available to the processors at timestep $t+1$. If $|T_t| > 0$ we call t a *queuing step*.

We call u the *last parent* of v in a schedule, if it is the last of v's parents to be completed in the schedule. Due to latencies on edges into v, it may not become ready until several timesteps after u is completed. For any task $v \in V$, let $q(v)$ be the number of queuing steps that take place after the last parent of v has been completed and before task v is ready. We define the *queuing delay* δ of a Q-prioritized schedule as $\delta = \max_{v \in V}\{q(v)\}$. This metric reflects the asynchronous overlap between the execution of the scheduling algorithm that inserts tasks into the work queue, and the parallel computation, and depends on how often the scheduling algorithm executes queuing steps.

Space bounds for Q-prioritized schedules. Let the *maximum task space* M of a task graph $G_T = (V, E)$ be the maximum over the memory requirements of all tasks in G_T; that is, $M = \max_{v \in V}\{h(v)\}$.

Theorem 6.1 *Let G_T be a task graph with depth d and maximum task space M. Let $\mathcal{S}_p$ be any Q-prioritized p-schedule for G_T based on any 1-schedule $\mathcal{S}_1$ of G_T. If $\mathcal{S}_p$ is generated using a work queue of size q_{max}, and δ is the queuing delay of $\mathcal{S}_p$, then space$(\mathcal{S}_p) \leq$ space$(\mathcal{S}_1) + ((\delta+1) \cdot q_{max} + p - 1) \cdot M \cdot d$.*

Proof: The proof can be found in [29][3]. It is based on showing that no more than $((\delta+1) \cdot q_{max} + p - 1)d$ tasks are scheduled out-of-order with respect to the serial schedule. These tasks, each of which has a memory requirement of at most M, result in the parallel execution requiring more space than the serial execution. ∎

We can now bound the space requirements of the schedules generated by algorithm Async-Q, by showing that they are Q-prioritized schedules.

Lemma 6.2 *For a parallel computation with w work, σ synchronizations, and d depth, in which every action allocates at most a constant K units of space, the Async-Q algorithm on p processors, with $q_{max} = p$, creates a task graph G_T with work $O(w)$, at most σ synchronizations, depth $O(d)$, latency-weighted depth $d_l = O((\sigma/p + d)\log l_m)$, and maximum task space $\log(pd)$, where l_m is the maximum number of live threads in L. For a constant α, $0 < \alpha < 1$, the algorithm generates a Q-prioritized $((1-\alpha)p)$-schedule for G_T based on the serial depth-first schedule, with a queuing delay of at most 1.*

Proof: As described in Section 5, algorithm Async-Q splits each thread into a series of tasks at runtime. Since each thread is executed non-preemptively as long as it does not synchronize, terminate, fork, or allocate more than a maximum of $M = \log(pd)$ units of memory, each resulting task v has a memory requirement $h(v) \leq \log(pd)$. For each task v in G_T, a processor performs two unit-time accesses to the queues Q_{out} and Q_{in}. Thus the total work performed by the processor for task v is $t(v) + 2$, where $t(v) \geq 1$ and $\sum t(v) = w$. Therefore the total work of G_T is $O(w)$, and similarly, the depth is $O(d)$. G_T has at most σ synchronizations, since besides forks, only the pairs of reads and writes that result in the suspension of the reading thread contribute to synchronizations in G_T.

[3] We use "queuing frequency" instead of "queuing delay" in [29].

Next, we show that the algorithm Async-Q generates Q-prioritized schedules with a queuing delay $\delta \leq 1$. If r is the number of ready threads in R, the scheduler puts $\min(r, p - |Q_{out}|)$ tasks into Q_{out}. Moreover, these tasks are the tasks with the highest priorities in R, where the priorities are based on the serial, depth-first execution order. Therefore, using Q_{out} as the FIFO work queue of maximum size $q_{max} = p$, and the ordered set R to represent the set of ready tasks that are not in the work queue, Async-Q generates a Q-prioritized schedule. With $(1-\alpha)p$ processors executing the worker computation, the resulting schedule is a Q-prioritized $((1-\alpha)p)$-schedule. The last timestep of every scheduling iteration is a queuing step, since all the tasks moved to Q_{out} at the end of that iteration are available after this timestep. Consider any task v in G_T. Let t be the timestep in which the last parent u of v is completed. u is placed in Q_{in} at timestep $t+1$ (assuming the insertion uses a unit-time fetch-and-add). In the worst case, a scheduling iteration may end in timestep $t+1$, making $t+1$ a queuing step. However, the next scheduling iteration must find u in Q_{in}. Since u was the last parent of v, v becomes ready during this scheduling iteration (we consider it to become ready just before the last timestep of the iteration). Therefore, the next queuing step, which is the last timestep of this scheduling iteration, takes place after v becomes ready. Thus, for any v, at most one queuing step can take place after its last parent is completed and before v becomes ready, that is, $\delta \leq 1$.

Finally, we bound the latency-weighted depth d_l of G_T. Consider any path in G_T. Let l be its length. For any edge (u, v) along the path, if u is the last parent of v, then v becomes ready at the end of at most two scheduling iterations after u is computed. Therefore the latency $l(u, v)$ is at most the durations of two scheduling iterations[4]. Since any path in G_T has at most $(d-1)$ edges, the latency-weighted depth of the path is at most the sum of the times required for $2(d-1)$ scheduling iterations plus the depth d, which is, using Lemma 5.2, $O((\sigma/p + d)\log l_m)$. ∎

Next, we bound the size of L.

Lemma 6.3 *The maximum number of live threads in L is $l_m = O((q_{max} + p)d)$, which is $O(pd)$ when $q_{max} = p$.*

Proof: The maximum number of live threads during the execution of a Q-prioritized p-schedule $\mathcal{S}_p$ (with queuing delay δ) based on $\mathcal{S}_1$ exceeds the maximum number of live threads during the execution of $\mathcal{S}_1$ by at most $((\delta+1) \cdot q_{max} + p - 1) \cdot O(d) = O((q_{max} + p)d)$ (using Lemma 6.2 and Lemma 4.1 in [29]), plus the number of threads in Q_{out} (which may have been created but have not started execution yet). Since $\mathcal{S}_1$ is a depth-first schedule, at most d threads can exist during its execution. Further, Q_{out} can have at most q_{max} threads. Therefore, L, which contains all the live threads, can have at most $O((q_{max} + p)d)$ threads. ∎

We can now bound the number of timesteps required to execute the resulting schedules.

Lemma 6.4 *Let G_T be the task graph created by algorithm Async-Q for a parallel computation with w work, σ synchronizations, and d depth, and let $\mathcal{S}_p$ be the $(1-\alpha)p$-schedule generated for G_T, where α is a constant $(0 < \alpha < 1)$. If $q_{max} = p$, then the length of $\mathcal{S}_p$ is $|\mathcal{S}_p| = O((w + \sigma\log(pd))/p + d\log(pd))$.*

[4] If u is not the last parent of v, we can use $l(u, v) = 0$ since it does not affect the schedule or its analysis.

Proof: We will show that $\mathcal{S}_p$ is a greedy schedule, with $O((w+\sigma\log(pd))/p)$ additional timesteps in which the workers may be idle. Consider any scheduling iteration. Let t_i be the timestep at which the i^{th} scheduling iteration ends. After tasks are inserted into Q_{out} by the i^{th} scheduling iteration, there are two possibilities:

1. $|Q_{out}| < p$. This implies that all the ready tasks are in Q_{out}, and no new tasks become ready until the end of the next scheduling iteration. Therefore, at every timestep j such that $t_i < j \leq t_{i+1}$, if m_j processors become idle and r_j tasks are ready, $\min(m_j, r_j)$ tasks are scheduled.

2. $|Q_{out}| = p$. Since $(1-\alpha)p$ worker processors will require at least $1/(1-\alpha)$ timesteps to execute p tasks, none of the processors will be idle for the first $1/(1-\alpha) = O(1)$ steps after t_i. However, if the $(i+1)^{th}$ scheduling iteration, which is currently executing, has to awaken n_{i+1} suspended threads, it may execute for $O((n_{i+1}/p+1)\log(pd))$ timesteps (using Lemmas 5.2 and 6.3). Therefore, some or all of the worker processors may remain idle for $O((n_{i+1}/p+1)\log(pd))$ timesteps before the next scheduling step; we call such steps *idling timesteps*. We split the idling timesteps of each scheduling iteration into the first $I_1 = \Theta(\log(pd))$ idling timesteps, and the remaining I_2 idling timesteps. A task with a fork or data edge out of it may execute for less than I_1 timesteps; we call such tasks *synchronization tasks*. However, all other tasks, called *thread tasks*, must execute for at least I_1 timesteps, since they execute until their space requirement reaches $\log(pd)$, and every action may allocate at most a constant amount of memory. Therefore, if out of the p tasks on Q_{out}, p_σ are synchronization tasks, then during the first I_1 steps of the iteration, at most p_σ processors will be idle, while the rest are busy. This is equivalent to keeping these p_σ processors "busy" executing no-ops (dummy work) during the first I_1 idling timesteps. Since there are at most σ synchronization tasks, this is equivalent to adding $\sigma\log(pd)$ no-ops, increasing the work in G_T to $w' = O(w+\sigma\log(pd))$, and increasing its latency-weighted depth d_l by an additive factor of at most $\Theta(\log(pd))$. There can be at most $O(w'/p) = O(w+\sigma\log(pd))/p$ such steps in which all worker processors are "busy". Therefore, the I_1 idling timesteps in each scheduling iteration can add up to at most $O((w+\sigma\log(pd))/p)$. Further, since a total of $O(\sigma)$ suspended threads may be awakened, if the $(i+1)^{th}$ scheduling iteration results in an additional $I_2 = O(n_{i+1}\log(pd)/p)$ idling timesteps, they can add up to at most $O(\sigma\log(pd)/p)$. Therefore, a total of $O((w+\sigma\log(pd))/p)$ idling timesteps can result due the scheduler.

All timesteps besides the idling timesteps caused by the scheduler obey the conditions required to make it a greedy schedule, and therefore add up to $O(w')/p + d_l = O((w+\sigma\log(pd))/p + d\log(pd))$ (using Lemmas 4.1 and 6.2). Along with the idling timesteps, the schedule requires a total of $O((w+\sigma\log(pd))/p + d\log(pd))$ timesteps. ∎

Note that since $q_{max} = p$, the maximum number of threads in both Q_{in} and Q_{out} is $O(p)$, and each thread can be represented using constant space[5]. Therefore, using Theorem 6.1 and Lemmas 6.2, 6.3, and 6.4 we obtain the following theorem which includes scheduler overheads.

[5] This is the memory required to store its state such as registers, not including the stack and heap data.

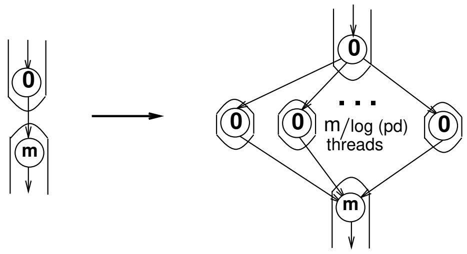

Figure 7: A transformation of the parallel computation to handle a large allocation of space within an action without violating the space bound. Consecutive actions of a thread are grouped into tasks, shown as outlines around the actions (nodes). Each action is labeled with the amount of memory its action allocates.

Theorem 6.5 *For a parallel computation with w work, σ synchronizations, d depth, and s_1 sequential space, in which at most a constant amount of memory is allocated in each action, the Async-Q algorithm (with $q_{max} = p$) generates a schedule for the parallel computation and executes it on p processors in $O((w+\sigma\log(pd))/p + d\log(pd))$ time steps, requiring a total of $s_1 + O(dp\log(pd))$ units of memory.* ∎

Handling arbitrarily big allocations. Actions that allocate more that a constant K units of memory are handled in the following manner, similar to the technique suggested in [4] and [29]. The key idea is to delay the big allocations, so that if tasks with higher priorities become ready, they will be executed instead. Consider an action in a thread that allocates m units of space ($m > K$), in a parallel computation with work w and depth d. We transform the computation by inserting a fork of $m/\log(pd)$ parallel threads before the memory allocation (see Figure 7). These new child threads do not allocate any space, but each of them perform a dummy task of $\log(pd)$ units of work (no-ops). By the time the last of these new threads gets executed, and the execution of the original parent thread is resumed, we have scheduled $m/\log(pd)$ tasks. These $m/\log(pd)$ tasks are allowed to allocate a total of m space, since we set the maximum task space $M = \log(pd)$. However, since they do not actually allocate any space, the original parent thread may now proceed with the allocation of m space without exceeding our space bound.

This transformation requires the scheduler to implement a fork that creates an arbitrary number of child threads. To prevent L from growing too large, the child threads are created lazily by the scheduler and a synchronization counter is required to synchronize them (see [29] for details). Let S_K be the *excess allocation* in the parallel computation, defined as the sum of all memory allocations greater than K units. Then the work in the transformed task graph is $O(w+S_K)$, and the number of synchronizations is $\sigma + 2S_K/\log(pd)$. As a result, the above time bound becomes $O((w+S_K+\sigma\log(pd))/p + d\log(pd))$, while the space bound remains unchanged. When $S_K = O(w)$, the time bound also remains unchanged.

Theorem 6.5 can now be generalized to allow arbitrarily big allocations of space. Note that the space and time bounds include the overheads of the scheduler.

Theorem 6.6 *Let $\mathcal{S}_1$ be the serial depth-first schedule for a parallel computation with w work, σ synchronizations, and d depth. For any constant $K \geq 1$, let S_K be the excess* *allocation in the parallel computation. The Async-Q algorithm, with $q_{max} = p \log p$, generates a schedule for the parallel computation and executes it on p processors in $O((w + S_K + \sigma \log(pd))/p + d \log(pd))$ time steps, requiring a total of space$(\mathcal{S}_1) + O(dp \log(pd))$ units of memory.* ∎

Remark. If the depth d of the parallel computation is not known at runtime, suspending the current thread just before the memory requirement exceeds $\log(pd)$ units is not possible. Instead, if L contains l threads when a thread is put into Q_{out}, setting its maximum memory to $O(\log(l+p))$ units results in the same space and time bounds as above.

7 Optimal scheduling of planar computation graphs

In this section, we provide a work- and space-efficient scheduling algorithm, denoted Planar Asynch-Q, for planar computation DAGs. The main result used by the algorithm is a theorem showing that, for planar DAGs, the children of any node v have the same relative priority order as v; this greatly simplifies the task of maintaining the ready nodes in priority order at each scheduling iteration. We conclude this section by sketching an algorithm that is a hybrid of Planar Asynch-Q and Asynch-Q, suitable for general DAGs.

Maintaining priority order for planar graphs. For a computation graph G, the following general scheduling algorithm maintains the set, R^*, of its ready nodes (actions) in priority order according to the 1DF-schedule $\mathcal{S}_1$.

Algorithm Planar:
R^* is an ordered set of ready nodes initialized to the root of G. Repeat at every timestep 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^*.

Note that Algorithm Planar does not require the subset scheduled in step 1 to be the highest-priority nodes in R^*. Moreover, it does not maintain in R^* place-holders for suspended nodes in order to remember their priorities. Instead, each newly-reactivated suspended node will be inserted into R^* (in step 2) in the place where the node activating it was, since it is a child of its activating node. We show below that for planar computation graphs, priority order is maintained. In [4], we showed that a similar stack-based scheduling algorithm, where we restrict step 1 to schedule only the highest-priority nodes, can be used to maintain priority order for any series-parallel computation graph. The proof was fairly straight-forward due to the highly-structured nature of series-parallel graphs; it relied on properties of series-parallel graphs not true for planar graphs in general. Thus for the following theorem, we have developed an entirely new proof (the precise definitions and the proof are given in Appendix A).

Theorem 7.1 *For any single root s, single leaf t, (s,t)-planar computation graph G with counterclockwise edge priorities, the online Algorithm Planar above maintains the set R^*, of ready nodes in priority order according to the 1DF-schedule $\mathcal{S}_1$.* ∎

Left-to-right synchronization edges. In the remainder of this section, we consider an important class of DAGs such that the write of any synchronization variable precedes any read of the variable when the computation is executed according to a serial depth-first schedule. In languages with futures, for example, this implies that in the serial schedule, the part of the computation that computes the futures value precedes the part of the computation that uses the futures value. We refer to such DAGs as having *left-to-right synchronization edges*; and example is given in Figure 1.

Implementing the scheduler for planar computation graphs. We next show how Algorithm Planar can be used as the basis for an asynchronous, non-preemptive scheduler that uses tasks as the unit of scheduling, for planar DAGs with left-to-right synchronization edges. We modify the scheduler for algorithm Async-Q as follows. Instead of maintaining the prioritized set of all live threads L, the scheduler maintains the prioritized set R^*, which contains the ready and active threads. Suspended threads are queued up in the synchronization queue for their respective synchronization variable, but are not kept in R^*. Since there are no suspended threads in R^*, techniques developed previously [29] for programs without synchronization variables can be used to obtain our desired bounds, specifically, an array implementation that uses lazy forking and deleting with suitable prefix-sums operations.

When a thread writes to a synchronization variable, it checks the synchronization queue for the variable, and awakens any thread in the queue. In an (s,t)-planar DAG with left-to-right synchronization edges, there can be at most one suspended reader awaiting the writer of a synchronization variable. (Any such reader must have at least two parents: the writer w and some node that is not a descendant of w or any other reader. A simple argument shows that for the DAG to be planar, there can be at most one such reader to the "right" of w.) Thus fetch-and-add is not needed for the synchronization queues, and in fact an EREW PRAM suffices to implement the scheduler processors. Following Algorithm Planar, we insert the suspended thread just after the writer thread in R^*, thereby maintaining the priority order.

At each scheduling iteration, the scheduler processors append to Q_{out} the $\min(|R|, q_{max} - |Q_{out}|)$ ready threads with highest priority in R^*. The worker processors will select threads from the head of Q_{out} using a fetch-and-add primitive. Denoting the modified Async-Q algorithm as *Planar Async-Q*, we have the following theorem:

Theorem 7.2 *Let $\mathcal{S}_1$ be the 1DF-schedule for a parallel computation with synchronization variables that has w work, d depth, at most a constant amount of memory allocated in each action, and whose computation graph is (s,t)-planar with counterclockwise edge priorities and left-to-right synchronization edges. The Planar Async-Q algorithm, with $q_{max} = p \log p$, generates a schedule for the parallel computation and executes it on p processors in $O(w/p + d \log p)$ time steps, requiring a total of space$(\mathcal{S}_1) + O(dp \log p)$ units of memory. The scheduler processors run on an EREW PRAM; the worker processors employ a constant-time fetch-and-add primitive.* ∎

A hybrid algorithm. In general, it is not known a priori if the computation graph is planar. Thus in the full paper, we develop a hybrid of Async-Q and Planar Async-Q that works for any parallel program with synchronization variables, and runs within the time and space bounds for the planar algorithm if the computation graph is planar or near planar, and otherwise runs within the bounds for the general algorithm. The hybrid algorithm starts by running a slightly modified Planar Async-Q algorithm which maintains, for each node v in R^*, a linked list of the suspended nodes priority ordered after v and before the next node in R^*. By Lemma 6.3, we know that the number of suspended nodes is $O(pd \log p)$, and we allocate list items from a block of memory of that size. As long as any node that writes a synchronization variable reactivates the first suspended node in its list, as will be the case for planar computation graphs with left-to-right synchronization edges and possibly others, the hybrid algorithm continues with this approach. When this is not the case, then we switch to the (general) Async-Q algorithm. The set L needed for algorithm Async-Q is simply the set of threads corresponding to nodes in R^* and in the suspended nodes lists. From R^* and the suspended nodes lists, we link up one long list of all threads in L in priority order. Since all linked list items have been allocated from a contiguous block of memory of size $O(dp \log p)$, we can perform list ranking to number the entire list in order and then create a black-white priority queue as a balanced binary tree. We can then proceed with the Async-Q algorithm.

8 Discussion

Here we mention some issues concerning the practicality of the technique. First we note that although the implementation uses fetch-and-add, the only places where it is used are for the processors to access the work queues (in which case we can get away with a small constant number of variables), and to handle the queues of suspended jobs. Other work [6] has shown that for certain types of code the number of reads to any synchronization variable can be limited to one, making the fetch-and-add unnecessary for handling the queues of suspended jobs.

If the parallel computation is very fine grained, the number of synchronizations σ can be as large as the work w, resulting in a running time of $O(\log(pd) \cdot (w/p + d))$, which is not work efficient. However, since synchronizations are expensive in any implementation, there has been considerable work in reducing the number of synchronizations using compile-time analysis [19, 15, 35, 36, 25]. We plan to explore the use of such methods to improve the running time of our implementation.

The implementation described for the scheduling algorithm assumes that a constant fraction of the processors are assigned to the scheduler computation, eliminating them from the work-force of the computational tasks. An alternative approach is to have all processors serve as workers, and assign to the scheduler computation only processors that are idle, between putting their thread in Q_{in}, and taking their new threads from Q_{out}. (Details will be given in the full paper.)

We finally remark that the various queues used in the scheduling algorithm can be implemented using asynchronous low-contention data structures such as counting networks [2] and diffracting trees [37].

References

[1] 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.

[2] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

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

[4] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[5] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proc. International Conference on Functional Programming*, May 1996.

[6] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. Proc. Symposium on Parallel Algorithms and Architectures, June 1997.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiseron, K. H. Randall, and Zhou Yuli. CILK: an efficient multithreaded runtime system. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

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

[9] 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.

[10] 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.

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

[12] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In David Padua, David Gelernter, and Alexandru Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 95–113. The MIT Press, 1990.

[13] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with OLDEN (parallel programming). In *Proc. International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, August 1993.

[14] R. Chandra, A. Gupta, and J. Hennessy. COOL: A Language for Parallel Programming. In David Padua, David Gelernter, and Alexandru Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 126–148. The MIT Press, 1990.

[15] C. D. Clack and S. L. Peyton Jones. Strictness analysis – a practical approach. In *Proc. Functional Programminng Languages and Computer Architecture*. Springer-Verlag LNCS 201, Sept. 1985.

[16] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proc. Intl. Symposium on Computer Architecture*, pages 141–150, May 1988.

[17] E.G. Coffman, Jr., editor. *Computer and job-shop scheduling theory.* John Wiley & Sons, New York, 1976.

[18] 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.

[19] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimizations. In *Proc. Symposium on Principles of Programming Languages*, pages 209–220, Jan. 1995.

[20] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

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

[22] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, January 1996.

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

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

[25] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time partitioning of a non-strict language into sequential threads. In *Proc. Symposium on Parallel and Distributed Computing*, Dec. 1991.

[26] 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.

[27] D. A. Krantz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proc. Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[28] 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.

[29] G. J. Narlikar and G. E. Blelloch. A framework for space and time efficient scheduling of parallelism. Technical Report CMU-CS-96-197, Computer Science Department, Carnegie Mellon University, 1996.

[30] G. J. Narlikar and G. E. Blelloch. Space-efficient implementation of nested parallelism. In *Proc. Symposium on Principles and Practice of Parallel Programming*, June 1997.

[31] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Lecture Notes in Computer Science 143: Proc. Colloquium on Automata, Languages and Programming, Barcelona, Spain*, pages 597–609, Berlin/New York, July 1983. Springer-Verlag.

[32] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[33] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, June 1993.

[34] 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.

[35] K. E. Schauser. Compiling lenient languages for parallel asynchronous execution. Technical Report UCB/CSD 94/832, University of California, Berkeley, 1994.

[36] K.E. Schauser, D. E. Culler, and S. C. Goldstein. Separation constraint partitioning-a new algorithm for partitioning non-strict programs into sequential threads. In *Proc. Symposium on Principles of Programming Languages*, pages 259–71, Jan. 1995.

[37] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. In *Proc. Symposium on Parallel Algorithms and Architectures*, pages 33–41, Padua, June 1996. ACM.

[38] B. T. Smith. Parallel computing forum (PCF) fortran. In *Aspects of Computation on Asynchronous Parallel Processors. Proc. IFIP WG 2.5 Working Conference.* North-Holland, August 1988.

A Further details on planar graphs

Definitions. 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. To define a 1-schedule for G 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 A.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 $\mathcal{S}_1$) 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 A.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. If actions of a DAG are numbered in the order in which they appear in a 1DF-schedule, we call the resulting numbers the 1DF-*numbers* of the actions.

Theorem 7.1 *For any single root s, single leaf t, (s,t)-planar computation graph G with counterclockwise edge priorities, the online Algorithm Planar above maintains the set R^*, of ready nodes in priority-order according to the* 1DF-*schedule $\mathcal{S}_1$.*

Proof: We first prove properties about the 1DF-numbering of G, and then use these properties to argue Algorithm Planar 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 Algorithm Planar 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 Algorithm Planar schedules a subset of R^* and puts ready children back in place it maintains R^* ordered relative to the 1DF-numbering. ∎

Note that the previous theorem held for any planar computation graph, with arbitrary fan-in and fan-out, and did not use properties of computations with synchronization variables.