Low-Contention Depth-First Scheduling of Parallel Computations with Write-Once Synchronization Variables*

Panagiota Fatourou

Department of Computer Science, University of Toronto
`faturu@cs.toronto.edu`

Abstract

We present an efficient, randomized, online, scheduling algorithm for a large class of programs with write-once synchronization variables. The algorithm combines the *work-stealing* paradigm with the *depth-first* scheduling technique, resulting in high space efficiency and good time complexity. By automatically increasing the *granularity* of the work scheduled on each processor, our algorithm achieves good locality, low contention and low scheduling overhead, improving upon a previous depth-first scheduling algorithm [6] published in SPAA'97. Moreover, it is provably efficient for the general class of multithreaded computations with write-once synchronization variables (as studied in [6]), improving upon algorithm `DFDeques` (published in SPAA'99 [24]), which is only for the more restricted class of *nested parallel* computations.

More specifically, consider such a computation with *work* T_1, *depth* T_∞ and σ *synchronizations*, and suppose that space S_1 suffices to execute the computation on a single-processor computer. Then, on a P-processor shared-memory parallel machine, the expected space complexity of our algorithm is at most $S_1 + O(PT_\infty \log(PT_\infty))$, and its expected time complexity is $O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$. Moreover, for any $\epsilon > 0$, the space complexity of our algorithm is $S_1 + O(P(T_\infty + \ln(1/\epsilon))\log(P(T_\infty + \ln(P(T_\infty + \ln(1/\epsilon))/\epsilon))))$ with probability at least $1-\epsilon$. Thus, even for values of ϵ as small as e^{-T_∞}, the space complexity of our algorithm is at most $S_1 + O(PT_\infty \log(PT_\infty))$ with probability at least $1 - e^{-T_\infty}$. These bounds include all time and space costs for both the computation and the scheduler.

1 Introduction

Many parallel programming languages [3, 7, 11, 12, 14, 18, 20, 22, 28] support dynamic threads. The multithreaded model of parallel computation is a general approach to model dynamic, unstructured parallelism. During the execution of a multithreaded computation, a thread may spawn child threads which can be executed in parallel, and it can synchronize with other currently executing threads. In most of the work in the literature [1, 4, 5, 6, 7, 9, 15, 16, 24, 25, 26, 27], a multithreaded computation is modeled as a directed acyclic graph (see Figure 1(a)).

Of much concern is how a multithreaded computation can be executed efficiently on a parallel computer. A parallel execution of a multithreaded computation specifies which processor executes each thread and when each thread is executed. A good scheduler must ensure that enough threads remain active to keep processors busy. On the other hand, the scheduler may need to limit the number of concurrently active threads to guarantee good space complexity. This is very important because parallel programs usually solve large problems and limitations on the available memory lead often to performance degradation. Another important objective of an efficient scheduler is to obtain good locality, which is usually achieved by scheduling threads close in the computation graph (that is, threads that satisfy the ancestor-descendant relationship) on the same processor. Designing a scheduling algorithm to achieve all of the above goals is not a trivial task.

Several algorithms [7, 9, 15, 16] employ *work-stealing,* a technique in which underutilized processors try to "steal" work from over-utilized ones, to achieve the above scheduling goals. Recently, a flurry of research [4, 6, 24, 26] has resulted in *depth-first* schedulers, which schedule threads prioritized by their (serial) "left-to-right depth-first" execution order and are highly space efficient; the *space complexity* of an algorithm is the total amount of memory used by all processors to execute the computation. Depth-first schedulers achieve space complexity $S_1 + O(PT_\infty)$, where S_1 is the *optimal* space needed to execute the computation on a single processor, T_1 is the *work* of the computation, that is the total number of its instructions, and T_∞ is its *dag depth*, that is the maximum length of any path in the computation graph. For computations with sufficient parallelism, depth-first schedulers improve upon the previous space bound of $O(PS_1)$ achieved by work-stealing algorithms [7, 9, 15, 16]. Moreover, depth-first schedulers cope with heap allocation [5] which is more general than the stack-based model assumed in work on work-stealing [7, 8, 9, 15, 16]. However, depth-first schedulers use a globally ordered centralized data structure of "active" threads and thus they are not as practical as work-stealing schedulers. Especially for fine-grained computations, they may perform poorly due to low locality and high scheduling contention for accessing shared data struc-

*This work was done while the author was affiliated with the Max-Planck Institute für Informatik, Saarbrücken, Germany and was partially supported by the IST Programme of the European Union under contract number IST-1999-14186 (ALCOM-FT).

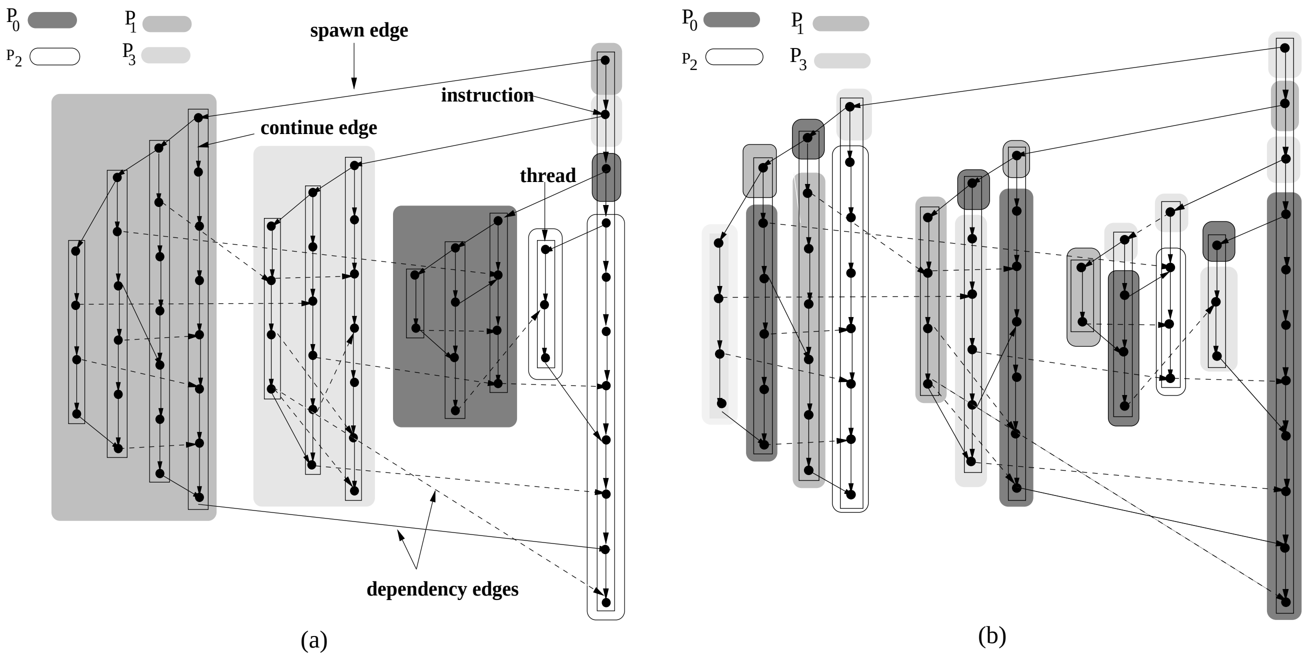

Figure 1: Different mappings of the threads of a multithreaded computation on four processors. Each one of figures (a) and (b) represent the computation graph of the same multithreaded computation mapped on four parallel processors by two different algorithms. Circles represent serially-executed, unit-time instructions, while the boxes of vertical chains of circles represent threads. The vertical edges, called *continue edges*, determine the execution ordering of the instructions of a thread, the right-to-left edges, called *spawn edges*, represent spawning of new threads which can be executed in parallel with each other and with the parent thread, while accesses of different threads to shared (synchronization) variables is represented by left-to-right edges, called *dependency edges*. A dependency edge exists from any instruction that writes a variable to all instructions that read it. Threads belonging to the same rounded box are considered to be "close" in the computation graph. Figures (a) and (b) illustrate the scheduling granularities of two different algorithms. The rounded boxes represent how threads are mapped onto processors. Clearly, the first mapping provides larger scheduling granularity, and thus better cache locality and lower contention.

tures. Narlikar [24] defines the *scheduling granularity* to be the average number of instructions executed consecutively on a single processor from threads close together in the computation graph. A larger scheduling granularity implies better locality and lower contention. Figure 1 represents two different mappings of the threads of some computation onto processors. Clearly, the mapping of Figure 1(a) is significantly better in terms of locality and contention than the one of Figure 1(b).

In this work, we present a new scheduling algorithm which is space and time efficient for computations with write-once synchronization variables, that is, variables which can be written by only one thread and read by any number of other threads. This class of computations, introduced in [6], is much more general than the *nested parallel* or *fully-strict* computations, where threads synchronize only with their parents. These last classes of computations do not include e.g., parallel programs using futures [11, 20, 22], lenient or speculative evaluation [2] or general user-specified synchronization constraints [28], which can be implemented using write-once synchronization variables [6]. Our algorithm is randomized and employs the work-stealing paradigm, combined with the depth-first technique. The algorithm not only exhibits the good space-complexity of depth-first schedulers, but also has good scheduling granularity and thus exhibits good locality and low contention.

Our algorithm works in the spirit of the algorithm presented in [24], which performs well only for nested-parallel computations. However, in order to cope with the more general class of computations with synchronization variables, our algorithm employs more complicated data structures. Our algorithm uses a 2–3 parallel priority tree data structure [6, 21] and stores at its leaves lists of threads, which are globally ordered according to their depth-first execution order. Each processor owns one of these lists and works on it as long as there are ready threads in it. When a processor becomes idle it tries to steal the topmost thread of a list chosen uniformly at random from a set of high priority lists. Because of randomization, the size of the 2–3 priority tree is a random variable. The algorithm has good scheduling granularity since it schedules threads of the same ready list on the same processor; our algorithm is likely to achieve the mapping provided in Figure 1(a). Moreover, each stolen thread is an ancestor of all other threads in its list, and thus it is the coarsest thread in the list. The processors executing the computation run asynchronously and they contend only when they try to steal from the same ready list as other processors. Our analysis significantly extends the space bound analysis presented in [24] to apply to the more general class of computations with write-once synchronization variables, if specific properties (arising from our analysis) hold. Scheduling computations with synchronization variables is more complicated because it allows for more general dependency edges; that is, a thread may synchronize with another thread which is not its parent or even its ancestor. For example, in Figure 1 all dashed edges represent more general synchronization constraints than those allowed by a nested-parallel or a fully-strict program. Our space bound analysis is much more complicated than the one presented in [24], because our algorithm employs more complicated data structures, as well as *variable memory quota* which are logarithmic functions of random variables. Proving our time bound employs new interesting techniques.

Blumofe and Leiserson [7, 9] were the first to prove that work-stealing algorithms are both space and time efficient for fully-strict multithreaded computations. Fatourou *et al.* have presented provably efficient scheduling algorithms for more general classes of multithreaded computations in [15, 16], generalizing the results presented in [7, 9] for programs that follow the general strict paradigm of parallelism; in a *strict* computation, a thread can synchronize with any of its ancestors, while in a fully-strict computation, a thread is allowed to synchronize only with its parent. Algorithms that achieve a space bound $O(PS_1)$ were also obtained by Burton [10], for a different class of parallel computations. A time-efficient algorithm for general computations which however ignores the space requirements, has been presented in [1].

The first depth-first schedulers were presented in [4, 26] for languages with nested fine-grained parallelism. These algorithms result in poor locality and low scheduling granularity, since threads close together in the computation graph are usually scheduled on different processors. Moreover, they suffer from high contention. It was very recently that Narlikar [24] presented algorithm `DFDeques` which allows processors to work on different (globally ordered) queues of ready threads. This results in better locality and lower scheduling contention. However, all this work has been limited to nested parallel computations. The only work that extends the depth-first approach to the more general class of computations with write-once synchronization variables is by Blelloch *et al.* [6]. However, their algorithm typically schedules close together threads on different processors, resulting in low scheduling granularity and suffering from high contention and bad locality. Our algorithm enjoys the good characteristics of low contention and good locality of `DFDeques` [24] but for the general class of computations with write-once synchronization variables studied by Blelloch *et al.* in [6]. Moreover, our algorithm is significantly better than the one in [6] in terms of good locality and low overheads due to contention; the granularity achieved by our scheduler is much better than the one of the algorithm in [6].

Although our algorithm works asynchronously, for the purpose of our analysis we assume, like in all previous work in this research area [1, 4, 5, 6, 7, 9, 15, 16, 24, 25], that the timesteps are synchronized across all the processors. In this work, we consider only deterministic parallel computations, that is, computations that always produce the same computation graph in every execution. Since the deterministic execution of parallel programs significantly simplifies their debugging, most parallel programming languages, even those that allow user-specified synchronization primitives, like, e.g., Jade [28], employ techniques for preserving the semantics of the corresponding serial programs. A very nice discussion about treating non-determinism under the multithreaded model is provided by Burton in [10, Section 6]. However, all known results [1, 4, 5, 6, 7, 9, 10, 15, 16, 24, 25] in this research area have been proved under the assumption of deterministic computations.

This paper is organized as follows. Section 2 presents our model. Our algorithm is presented in Section 3, while its analysis is discussed in Section 4.

2 Model

Our presentation closely follows those in [6, 24]. For each integer $n \geq 1$, let $[n]$ denote the set $\{1, \ldots, n\}$.

All threads share a single address space and execute a standard RAM instruction set augmented with instructions for (1) creating a new thread, (2) allocating and releasing a block of memory, (3) reading and writing to a local memory location, and (4) allocating, reading, writing and releasing a write-once synchronization variable. Each node γ that allocates (deallocates) M units of memory is assigned an integer weight $\mu(\gamma) = M$ ($\mu(\gamma) = -M$, respectively). For allocating a synchronization variable, $\mu(\gamma) = 1$. A *synchronization edge* is a spawn or a dependency edge; denote by σ the total number of synchronization edges of a computation. Recall that spawn edges represent spawning of new threads, while dependency edges exist in the computation graph from the instruction that writes a synchronization variable to the instructions that read it. A thread that performs a read on an unwritten synchronization variable *stalls* (or *suspends*). The thread is *enabled* when the variable is written by some other thread; we require that each such variable is written only once, and thus only one thread eventually writes the variable.

At any given timestep t, an unexecuted node is *ready* if all its predecessors in the instruction dag have been executed. A *P-processor execution schedule for a computation G* is a sequence $\mathcal{X}(G, P) = \mathcal{X}_1(G, P), \mathcal{X}_2(G, P), \ldots \mathcal{X}_T(G, P)$, of P-tuples such that (1) for each timestep $t \in [T]$, $\mathcal{X}_t(G, P) = \langle \gamma_1^t, \gamma_2^t, \ldots, \gamma_P^t \rangle$, where for each $i \in [P]$, γ_i^t, is the ready node executed by processor i at t, or $\emptyset$ if processor i does not execute any instruction at t, and (2) all nodes of G are contained in one of the $\mathcal{X}_t(G, P)$, $t \in [T]$. The space required by $\mathcal{X}(G, P)$, denoted $S(\mathcal{X}(G, P))$, is $S(\mathcal{X}(G, P)) = n + \max_{t=1,\ldots,T}\{\sum_{i=1}^{t} \sum_{\gamma \in \mathcal{X}_i(G,P) \& \gamma \neq \emptyset} \mu(\gamma)\}$, where n is the input data. The *length* of $\mathcal{X}(G, P)$, denoted by $|\mathcal{X}(G, P)|$, is the number of timesteps in it; that is, $|\mathcal{X}(G, P)| = T$.

During the execution of a computation, a thread is *active* from the timestep it is spawned until the timestep it executes all its instructions. If Γ is active at t, there exists some node $\gamma \in \Gamma$, called the *active* node of Γ at t, which is the *unique* ready or suspended node of Γ at t. For every edge $\langle \gamma', \gamma \rangle$ in the computation graph, we call node γ' a parent of node γ. Whenever a node γ' spawns a thread, γ' has two children. We say that its *left child* is the first node of the spawned thread, while its *right child* is its next node to be executed in its thread.

A depth-first multithreaded computation [7, Section 2.3] is a multithreaded computation in which the "left-to-right depth-first" search of nodes in the dag always visits all of any given node's predecessor in the dag before it visits the given node. More specifically, the *left-to-right depth first order* of nodes is defined [7, Section 2.3] as follows. If we ignore the dependency edges in the dag and we just consider the continue and spawn edges, we have a binary tree rooted at the first instruction of the root thread. The left-to-right depth-first order is then the order of nodes visited by a preorder walk of this tree [13]. Now considering the dependency edges again, the computation is depth-first if none of its dependency edges violate its left-to-right depth-first order. As in previous work on depth-first scheduling algorithms [6, 24, 25, 26], we concentrate on depth-first computations.

The left-to-right depth-first order yields a serial execution schedule, which we call the *DF-schedule* [4, 5, 6, 24] and we denote it by $\mathcal{X}_{DF}(G)$. The node γ scheduled at step

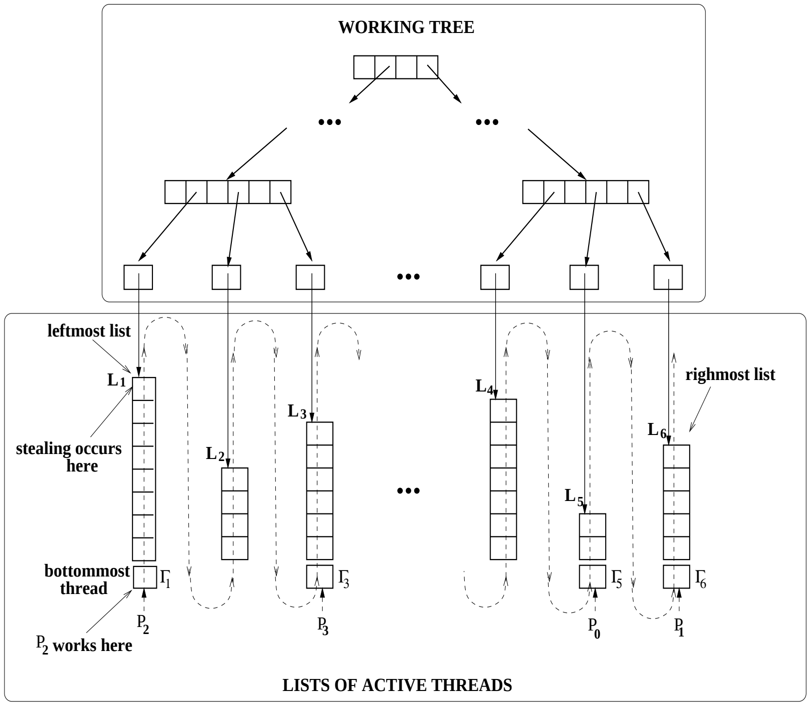

Figure 2: (a) The working tree and the lists of active threads maintained by DFPS. Notice that the leaves of the working tree contain pointers to the lists of active threads in the system. The dotted curved line represents the decreasing order of priorities of threads in the lists. We can see six lists in the system, namely L_1 to L_6. Only the lists L_1, L_3, L_5 and L_6 are owned by some processor (in our figure, these processors are p_2, p_3, p_0 and p_1, respectively), while all other lists are not owned by any processor. (b) The processors always work on the bottommost thread of each list. If some processor steals a thread out of some list, the stolen thread is the topmost thread of the list. Thus, by placing it in a newly created ready list on the immediate right of the victim list maintains the desired priority order. Additionally, if a thread stalls, it is the bottommost thread of its ready list (the one that the processor owning the list is working on). Thus, by placing it in a newly created list on the immediate left of the original list maintains again the desired priority order.

i, where $i \in [\,|\mathcal{X}_{DF}(G)|\,]$, is assigned priority $priority(\gamma) = T_1 - i$. The *priority of a thread* Γ at some timestep $t \in [\,|\mathcal{X}(G,P)|\,]$, denoted $priority_t(\Gamma)$, is defined to be the priority of the active node of Γ at t. Let $\langle \gamma_1, \ldots, \gamma_{|\mathcal{X}_{DF}(G)|} \rangle$ be the nodes of a computation G, in the order in which they are executed in $\mathcal{X}_{DF}(G)$. The *serial space requirement* [4, 5, 6, 24], denoted by S_1, is $S_1 = n + \max_{t=1,\ldots|\mathcal{X}_{DF}(G)|}\{\sum_{i=1}^{t}\mu(\gamma_i)\}$.

A *scheduling algorithm* for a multithreaded computation G decides which processors execute which instructions at every timestep. For any randomized scheduling algorithm Alg, let $T_{\texttt{Alg}}(G,P)$ and $S_{\texttt{Alg}}(G,P)$ be the random variables denoting the execution time (that is, the number of timesteps) and the total space required, respectively, during the P-processor execution of G by Alg.

3 The Algorithm

The algorithm, called DFPS (Depth-First Prioritizing Scheduler), classifies the processors into two categories, the *working processors* that execute the nodes of the computation, and the *scheduling processors* that manipulate the employed data structures. The scheduling processors consist of a constant fraction αP of the processors, where $0 < \alpha < 1$ is a constant chosen by the user. The remaining $P' = (1-\alpha)P$ processors are the working processors; all processors run in parallel.

All active threads are organized in a set of doubly-ended lists. Each list contains multiple threads with contiguous priorities. Thread priorities decrease from bottommost to topmost positions of each list; a list is prioritized according to the priority of its bottommost thread. A list can be in state *ready* or *stalled*, depending on the state of its bottommost thread. DFPS uses a 2-3 priority tree $\mathcal{B}$ [6, 21] of such lists, called the *working tree*. The lists reside at the leaves of the tree; all lists, independently of their states, are stored according to their priorities in $\mathcal{B}$. Thus, each leaf node of the working tree points to a list which is either ready or stalled. The priority of every thread in a list is larger than the priority of all the threads in lists to its right. Figure 2 (see part (a)) presents the working tree and the lists of active threads with their decreasing order of priorities, as maintained by DFPS. Instead of keeping keys, each internal node of the tree contains counters, which represent the number of ready lists in the subtree rooted at the node. The following operations, called the *priority tree operations*, can be applied to the priority tree: 1) given a set of leaf elements (that is, lists of threads): a) split each element belonging to the set into two adjacent elements; b) delete each element belonging to the set; c) change the state of each list belonging to the set from stalled to ready or vice versa, and 2) find the P leftmost *ready* lists in the working tree. A ready list may be owned by a working processor, but a processor

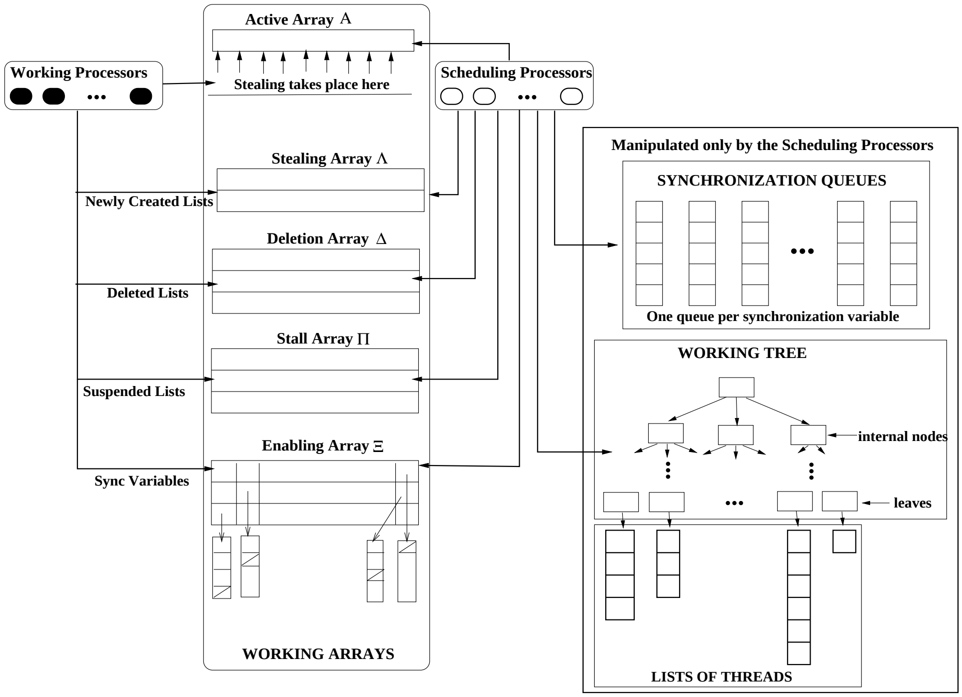

Figure 3: Data structures used by `DFPS`.

cannot own more than one lists.

The working processors do not explicitly access or update the working tree. It is only the scheduling processors that are allowed to do so. However, the working processors execute the computation, and thus they are the only processors that know any update which should be done to the working tree. On the other hand, the scheduling processors are the only to know which lists are the high priority lists in the tree. The working and the scheduling processors communicate by using a set of shared arrays, called the *working arrays*. The *active array* A is periodically updated to maintain pointers to the P' leftmost (that is, highest priority) ready lists in $\mathcal{B}$. The *deletion array* Δ contains pointers to lists that are to be deleted from $\mathcal{B}$, the *stall array* Π contains pointers to lists which contain suspended threads, while the *enabling array* Ξ contains pointers to lists of synchronization variables (whenever a working processor writes a variable with waiting threads, the processor adds a pointer to the variable in a list pointed to by some element of the enabling array). Finally, the algorithm uses the *stealing array* Λ to hold pointers to newly created lists that should be inserted in $\mathcal{B}$. The algorithm also uses a set of *synchronization queues* (studied in [19, Section 4.1]), one for each synchronization variable, in order to keep all suspended threads on each such variable. The elements of all the arrays are accessed periodically by the scheduling processors, so that appropriate modifications to the working tree, as well as to the synchronization queues are done. The working processors access only the working arrays, while the scheduling processors have access to all data structures. The working processors synchronize and communicate with the scheduling processors only through the set of arrays they share. Figure 3 presents all data structures used by the algorithm.

Initially, all the working processors are idle. An idle processor performs stealing in order to obtain a ready list. It selects an integer $m \in [P']$ uniformly at random and tries to steal the topmost thread from the ready list pointed to by the mth element of A, that is from the mth highest priority list. A steal attempt may fail for several reasons that we describe later in this section. If a steal attempt fails, the processor repeats its steal attempt by choosing another number $m \in [P']$ independently and uniformly at random, etc. If a processor's steal attempt is successful, the processor does the following. If the stolen thread is the last thread in the list and the list is not owned by any processor, the processor marks the target list as deleted. The stealing processor then creates a new ready list for itself, adds a pointer to this ready list in Λ, and starts executing the stolen thread. If the thread forks, p continues execution on the child thread, pushing back the parent thread on the bottommost position of its ready list. If the thread suspends, p marks its list as suspended by adding a pointer to the suspended list in Π, and starts work-stealing to obtain a new list. If the thread terminates, p removes the bottommost thread of its ready list and starts work on it. If the list is empty, p marks it as deleted by adding a pointer to the deleted list in Δ, and starts work-stealing. If p executes an instruction that writes on a synchronization variable with waiting threads, it adds a pointer to the synchronization variable, in one of the lists of synchronization variables pointed to by some element of Ξ.

Each working processor is allocated a memory quota of K bytes from which the threads in its ready list can allocate heap and stack data. Each time a processor steals, it is allocated an additional K bytes of memory quota. As in all depth-first schedulers, if a thread contains an action that needs more memory than the memory quota assigned to the ready list, e.g. the total memory required is $m > K$, $\lfloor m/K \rfloor$ dummy threads are inserted before the allocation. The dummy threads each performs a no-op operation, but

Successful Stealing: Actions taken by p

$h_p = m$;
if (target list now empty & not owned by some p')
mark the list as deleted;
create a new ready list L and become its owner;
add a pointer to L in $\Lambda[\lambda_{h_p}, h_p]$;
update λ_{h_p};
/* if ($\lambda_{h_p} == 1$) $\lambda_{h_p} = 2$; else $\lambda_{h_p} = 1$; */
start execution of the stolen thread;

Figure 4: Actions taken by a working processor p, whenever one of its steal attempts succeeds. We assume that p steals a thread from list A_m.

Marking a List as Suspended: Actions taken by p

if ($\Pi[3,p] == \perp$)
p inserts a pointer to the suspended list in $\Pi[3,p]$;
else
p inserts a pointer to the suspended list in $\Pi[\pi_{h_p}, h_p]$;
update π_{h_p};
/* if ($\pi_{h_p} == 1$) $\pi_{h_p} = 2$; else $\pi_{h_p} = 1$; */

Figure 5: Actions taken by a working processor p when it marks a list as suspended.

they do not allocate any space. Every time a processor executes a dummy thread, the processor gives up its ready list and performs stealing. Once all the dummy threads have been executed, the original thread can proceed with the large allocation of space. The addition of dummy threads effectively delays large allocations of space, so that higher priority threads may be scheduled instead.

As long as there exist threads in the system, the scheduling processors execute a loop containing the five tasks described below in the order given. We call each iteration of this loop a *scheduling iteration*.

(1) *Splitting Task*. For each element in the stealing array Λ, split the corresponding element of the working tree into two elements. The leftmost of these two elements points to the list pointed to by the initial element before the splitting, while the rightmost element points to the newly created list (see part (b) of Figure 2 for intuition). The counters of all its ancestors in the working tree are updated.

(2) *List Suspension Task*. For each element in Π, if the list pointed to by the element contains only the suspended thread, the list is marked as suspended, otherwise, the corresponding element of $\mathcal{B}$ is split into two elements. The leftmost element points to a newly created suspended list containing only the suspended thread, while the rightmost element points to the initial list (see part (b) of Figure 2 for intuition). In both cases, the counters of all ancestors in the working tree are updated. An element pointing to the suspended list is added to the synchronization queue of the corresponding synchronization variable.

(3) *Deletion Task*. For each element in Δ, the corresponding element of $\mathcal{B}$ is deleted.

(4) *Enabling Task*. All synchronization queues pointed to by elements of the lists of Ξ contain enabled threads. Each such thread is accessed and the state of the corresponding suspended list in $\mathcal{B}$ is modified from suspended to ready, while the counters of all its ancestors in the tree are updated.

(5) *Activation Task*. A's elements are updated to point to the P' leftmost ready lists in $\mathcal{B}$.

We allow only one steal attempt to take place at each ready list during a scheduling iteration. Thus, with each list, we associate a bit, called the *stealing bit*, which indicates whether or not stealing has been performed on the ready list. Every time a processor steals from a list it sets its stealing bit to 1. The stealing bits of lists that scheduling processors index during the activation task are reset to 0. All steal attempts to ready lists with stealing bit equal to 1 fail. If the number of ready lists is less than the size of the active array, all steal attempts directed to empty slots of the active array fail.

Consider any element e of some of the working arrays. We say that a scheduling iteration *updates* e if the scheduling processors process the information stored in e, and update the working tree according to this information. Assume that some scheduling iteration updates e at timestep t, while the next scheduling iteration updates e at timestep t'. Then, the time interval (t, t') is a *scheduling interval of* e.

We now present some technical details on the working arrays and the way the working processors manipulate them. All arrays have P' columns, one per working processor. The number of rows varies between one and three. The active array stores the P' highest priority lists of the working tree, and thus it has only one row. For each integer $m \in [P']$, it can be proved that at most two steal attempts targeting A_m can be successful during the scheduling interval $\mathcal{I}$ of an element of any of the other arrays; thus, at most $2P'$ lists are created during $\mathcal{I}$. Intuitively, this is so because we allow only one steal attempt to be successful on A_m during any scheduling iteration. Thus, two rows, of P' elements each, suffice to store all newly created lists during $\mathcal{I}$, so Λ has two rows. These newly created lists may have to be deleted or may suspend by the end of $\mathcal{I}$. In the worst case, a processor may have been working on an older list (from some previous scheduling interval) when $\mathcal{I}$ starts, and thus, each of the arrays Π and Δ should have three rows. The third of these rows contains default positions, one per working processor, where the working processors can store information about such older lists. The same is true for array Ξ.

Associated to each of the arrays Λ, Δ, Π, and Ξ are row *selection indices*, one for each column, denoted λ, δ, π, and ξ, respectively. Each selection index changes its value from 1 to 2 and vice versa every time it is accessed by some working processor; every time a selection index changes value, we say that it is *updated*. Initially, all arrays are empty and the value of all selection indices is 1. Each selection index is updated every time it is accessed by some working processor. Each working processor $p \in [P']$ maintains an integer h_p, called the *hitting index*, initially `NULL`. Every time a steal attempt by p on the list (pointed to by) A_m is successful, p stores m in its hitting index; that is, the hitting index contains the value of the index in A from which the processor obtained work most recently. Figure 4 summarizes the actions taken by a working processor, when one of its steal attempts succeeds.

The working processors manipulate arrays Π and Δ as follows. Each processor p has unique access to $\Pi[3,p]$ (or $\Delta[3,p]$ respectively); that is, an element of the third row of Π (Δ) is associated to p (this is the default element of p, used for storing information about "old" lists). Thus, each processor tries to store its data about suspended (deleted) lists in its associated element of Π (Δ). If it finds this element occupied, it then tries to write to different positions of the array. Then, it chooses the column of the array indexed by its hitting index and the row determined by the selection index of this column. Figure 5 summarize the actions taken by a working processor for marking a list as suspended. The actions taken for marking a list as deleted are similar. The enabling array Ξ is manipulated in a similar way. When a processor p executes an instruction that writes on a synchronization variable with waiting threads, p records the synchronization variable in the enabling array.

4 Analysis

Consider any multithreaded computation G and let $\mathcal{X}$ be an execution schedule produced by DFPS executing G on P processors. Denote $\mathcal{X}_{DF} = \mathcal{X}_{DF}(G)$. A *batch* of nodes is the set of all nodes executed by some processor between consecutive steal attempts of the processor. The first node executed in a batch is called *heavy*, while all other nodes of the batch are *light* nodes. Consider any timestep $t \in [\,|\mathcal{X}|\,]$. We define the *t-step prefix of* $\mathcal{X}$, denoted $\mathcal{X} \uparrow t$, to be the prefix of $\mathcal{X}$ consisting only of the first t timesteps. We define the *serial t-step prefix of* $\mathcal{X}$, denoted $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$, to be the longest prefix of $\mathcal{X}_{DF}$ such that each node of the prefix belongs in some $\mathcal{X}_i(G,P)$, $i \in [t]$. The *set of non-premature nodes at timestep t* are all nodes in $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$. The *set of premature nodes at timestep t* consists of all nodes γ such that $\gamma \notin (\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$, but there exists some integer $i \in [t]$ such that $\gamma \in \mathcal{X}_i(G,P)$.

For each node $\gamma \in G$ we define its *last parent* to be the last executed parent of γ. If there is more than one such node then the last parent of γ is found as follows. Let t be the timestep that these nodes are executed. If the thread containing γ belongs to (is placed in) the list of threads owned by the processor executing one of these parents or if it is executed at timestep $t+1$ by the processor executing one of these parents, then this parent is defined to be the last parent of γ; it is helpful later in our analysis to have γ and its last parent residing or being executed at the same processor (whenever possible). Otherwise, let the last parent of γ be any arbitrary node among its last executed parents.

We define the *critical path* of some node $\gamma \in G$, denoted P_γ, as follows. P_γ is initialized to γ; for each node $\gamma' \in P_\gamma$, the **last parent** of γ' is added to P_γ. Clearly, for each node γ, the root node must belong to P_γ; thus P_γ contains exactly all the nodes along a particular path from the root to γ. We denote by $\delta(\gamma)$ the depth of γ in this path. Let $P_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$. Denote by t_i the timestep that node γ_i is executed, $i \in [\delta(\gamma)]$. Since P_γ is a path, for each $i \in [\delta(\gamma)-1]$, the priority of node γ_i is larger than the priority of node γ_{i+1}. Thus, for each $i \in [\delta(\gamma)-1]$, $t_i < t_{i+1}$. For any timestep $t \in [|\mathcal{X}|]$ and any ready node $\gamma \in G$ at t, we say that t is a *candidate timestep for* γ, if the list of threads in which the thread containing γ belongs, is one of the target lists for steal attempts at t. For each $i \in [\delta(\gamma)]$, we denote by C_i the *set of candidate timesteps for* $\gamma_i \in P_\gamma$. Let $\mathcal{T}_\gamma = \{C_1, \ldots, C_{\delta(\gamma)}\}$; that is, $\mathcal{T}_\gamma$ is a set containing the sets of candidate timesteps of all nodes of P_γ.

If a number of steal attempts target the same list of threads at a particular timestep, we assume that only one of the steal attempts succeeds, while all the other fail in that timestep. If the ready list is empty, then all steal attempts fail in the timestep. When a steal attempt fails the processor tries again in the next timestep. Since at most P steal attempts can take place at each timestep, we can partition all timesteps in each of the C_i, where $i \in [\delta(\gamma)]$, $\gamma \in G$, into *rounds*, such that each round contains at least P and at most $2P-1$ successful steal attempts. Let $R = \langle t, \ldots, t' \rangle$ be any round. We say that R is *successful* if either node $\gamma' \in P_\gamma$, for which timestep t is a candidate timestep, is executed until timestep t', or one of the $\Theta(P)$ steal attempts initiated in R, targets the ready list containing γ'.

4.1 Properties of DFPS

We start by proving a set of properties for the working arrays. These properties guarantee good cooperation between the working and the scheduling processors. Let e be any element of some working array. We prove that e is written only once by the working processors during any of its scheduling intervals, so that information for updates on the working tree is never lost (e.g., because a working processor overwrites some of the elements of the working arrays). Thus, the working tree is always updated correctly by the scheduling processors.

Proposition 4.1 *For any arbitrary position e of some of the arrays Λ, Π, Δ or Ξ, e is written only once by the working processors during any of its scheduling intervals.*

Next, we prove that threads in any list of the working tree satisfy the parent-child relationship. We then show that threads in each list are in decreasing order of priority from bottom to top, while a currently executed thread on a processor has higher priority than all other threads on the processor's ready list. Moreover, we prove that the threads in any given list have higher priorities than all other threads in lists residing to the right of this list in the working tree. We use these properties to bound the number of heavy premature nodes in the system. Roughly speaking, for each timestep t of $\mathcal{X}$, we concentrate on the last executed non-premature node γ; that is, γ is the last executed node in $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$. If P_γ is the critical path of γ and t_γ is the timestep that γ is executed, then, for all timesteps $1, \ldots, t_\gamma$, exactly one of the nodes of P_γ is ready. Clearly, P_γ is a path from the root node to node γ, and thus its length cannot be more than T_∞. By using these facts we bound the number of rounds of steal attempts that occur until all of these nodes have been executed. It is then not difficult to bound the number of rounds of steal attempts that occur until timestep t. Since a heavy premature node is created only if a steal attempt is successful, we thus bound the number of heavy premature nodes in $\mathcal{X} \uparrow t$. More specifically, we prove the following two propositions, which we later use to bound the size of the working tree, as well as the space requirements of our algorithm.

Proposition 4.2 *Consider any integer $t \in [\,|\mathcal{X}|\,]$, let γ be the last node in $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$ and let t_γ be the timestep that γ is executed. Assume that $P_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of γ, $\mathcal{T}_\gamma$ is the set of sets of candidate timesteps of nodes of P_γ, and let $I'_\gamma = \{t_\gamma, \ldots, t\}$. Then,*

1. *the number of successful rounds in $\mathcal{T}_\gamma \cup I'_\gamma$ is $O(T_\infty)$,*
2. *the expected number of rounds in $\mathcal{T}_\gamma \cup I'_\gamma$ is $O(T_\infty)$,*
3. *for any $\varepsilon > 0$, the number of rounds in $\mathcal{T}_\gamma \cup I'_\gamma$ is $O(T_\infty + \ln(1/\varepsilon))$, with probability at least $1-\varepsilon$.*

Proposition 4.3 *Let γ be the last node in $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$ and let t_γ be the timestep that γ is executed. Assume that $P_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of γ, $\mathcal{T}_\gamma$ is the set* *of sets of candidate timesteps of nodes of P_γ, and $I'_\gamma = \{t_\gamma, \ldots, t\}$. Let R_t be the number of rounds of steal attempts in $\mathcal{T}_\gamma \cup I'_\gamma$. Then, the number of heavy premature nodes in $\mathcal{X} \uparrow t$ is $O(P(T_\infty + R_t))$.*

To bound the size of the working tree, we use the fact that all lists of threads in the system are created either by stealing or by suspension of threads. We bound the number of lists containing active threads with premature nodes at some timestep t by deriving a one-to-one correspondence between this set of lists and the set of heavy premature nodes at t. Bounding the number of lists with threads comprised only of non-premature nodes is based on the fact that the active threads in a serial execution cannot be more than T_∞.

Proposition 4.4 *Consider any integer $t \in [\,|\mathcal{X}|]$, let $|\mathcal{B}_t|$ be the size of the working tree at t, and let H_t be the number of heavy premature nodes at t. Then, $|\mathcal{B}_t| \leq 2(H_t + T_\infty)$.*

Thus, by Propositions 4.2, 4.3, and 4.4, it follows that: (1) $\mathcal{E}(|\mathcal{B}_t|) \in O(PT_\infty)$, and (2) for any $\epsilon > 0$, $|\mathcal{B}_t| \in O(P(T_\infty + \ln(1/\epsilon)))$, with probability at least $1 - \epsilon$.

4.2 Space Bound

Let $S^D_{\mathtt{DFPS}}(G, P)$ be the random variable denoting the memory used by our algorithm to store all the data structures it maintains. We prove that (1) $\mathcal{E}(S^D_{\mathtt{DFPS}}(G,P)) \in O(PT_\infty)$, and (2) for any $\varepsilon > 0$, $S^D_{\mathtt{DFPS}}(G,P) \in O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$.

When the memory-quota K is fixed during the execution, Propositions 4.2, 4.3, and 4.4 imply: (1) $\mathcal{E}(S_{\mathtt{DFPS}}(G,P)) \leq S_1 + O(PT_\infty K)$, and (2) for any $\epsilon > 0$, $S_{\mathtt{DFPS}}(G,P) \leq S_1 + O(P(T_\infty + \ln(1/\epsilon))K)$, with probability at least $1-\epsilon$. However, the choice of K influences not only the space complexity but also the time complexity of our algorithm. The best bound on the expected time complexity of our algorithm is achieved whenever K is variable and its value is reset to $\log(|\mathcal{B}_t| + P)$ at the beginning of each scheduling iteration, where t is the first timestep of the scheduling iteration and $|\mathcal{B}_t|$ is the size of the working tree at timestep t; $|\mathcal{B}_t|$ is a random variable, so that $\log(|\mathcal{B}_t| + P)$ is also a random variable. We use Jansen inequality (see e.g., [17]) to prove: (1) $\mathcal{E}(\log(|\mathcal{B}_t| + P)) \in O(\log(PT_\infty))$, and (2) for any $\epsilon > 0$, $\log(|\mathcal{B}_t| + P) \in O(\log(P(T_\infty + \ln(1/\epsilon)))$, with probability at least $1 - \epsilon$.

Theorem 4.5 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. Then,*

1. $\mathcal{E}(S_{\mathtt{DFPS}}(G,P)) \leq S_1 + O(PT_\infty \log(PT_\infty))$;
2. *for any $\epsilon > 0$, $S_{\mathtt{DFPS}}(G,P) \leq S_1 + O(P(T_\infty + \ln(1/\epsilon))$ $\log(P(T_\infty + \ln(P(T_\infty + \ln(1/\epsilon))/\epsilon))))$, with probability at least $1-\epsilon$.*

Sketch of proof: We account separately for the memory required by non-premature nodes and by premature nodes. The first class of nodes cannot allocate more than S_1 space, while all premature nodes are collected into batches, and all premature nodes of a batch are assigned to a heavy premature node. For the purpose of our analysis, we assume that all memory required by the premature nodes of a batch is allocated by the heavy premature node. Let S_p denote the memory allocated by the premature nodes, and let H_t be the random variable denoting the number of heavy premature nodes at t. For each such node γ_i, let t_i be the timestep at which the node is executed. Then, the node cannot allocate more than $K_{t_i} = \log(|\mathcal{B}_{t_i}| + P)$ memory. Thus, the total net memory allocated at t by premature nodes is $S_p = \sum_{i=1}^{H_t} \log(|\mathcal{B}_{t_i}| + P)$. In order to study $\mathcal{E}(S_p)$, we work with the random variable $S_p^{H_t} = \mathcal{E}(S_p \mid H_t)$, so that $S_p^{H_t}(n) = \mathcal{E}(S_p \mid H_t = n) = \mathcal{E}(\sum_{i=1}^n \log(|\mathcal{B}_{t_i}| + P)) = \sum_{i=1}^n \mathcal{E}(\log(|\mathcal{B}_{t_i}| + P))$. Recall that for each $t_i \in [\,|\mathcal{X}|]$, $\mathcal{E}(\log(|\mathcal{B}_{t_i}| + P)) \in O(\log(PT_\infty))$. Thus, there exists some constant $c > 0$ such that: $S_p^{H_t}(n) \leq c\sum_{i=1}^n \log(PT_\infty) = cn\log(PT_\infty)$. Clearly, $\mathcal{E}(S_p^{H_t}) = \mathcal{E}(\mathcal{E}(S_p \mid H_t))$, while it holds that $\mathcal{E}(S_p) = \mathcal{E}(\mathcal{E}(S_p \mid H_t))$ (see e.g., [23, Lemma 4.9, Chapter 4]). Thus,

$$
\begin{aligned}
\mathcal{E}(S_p) &= \sum_n S_p^{H_t}(n)\,\mathbf{Pr}[H_t = n] \\
&\leq \sum_n cn\log(PT_\infty)\,\mathbf{Pr}[H_t = n] \\
&= c\log(PT_\infty)\sum_n n\,\mathbf{Pr}[H_t = n] \\
&= c\log(PT_\infty)\,\mathcal{E}(H_t)\,.
\end{aligned}
$$

Propositions 4.2 and 4.3 imply that $\mathcal{E}(H_t) \in O(PT_\infty)$, so that $\mathcal{E}(S_p) \in O(PT_\infty \log(PT_\infty))$.

We continue to prove (2). For clarity of presentation, let $\beta_\varepsilon = (T_\infty + \ln(2/\varepsilon))$ and $\lambda_\varepsilon = \log(P(T_\infty + \ln(2c_1 P\beta_\varepsilon/\varepsilon))$. We prove that $\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon] \geq 1 - \varepsilon$, where $c_1, c_2 > 0$ are constants (to be determined later).

Let γ be the last non-premature node to be executed in $\mathcal{X} \uparrow t$; that is, γ is the last node in $(\mathcal{X} \uparrow t) \mid \mathcal{X}_{DF}$. Assume that t_γ is the timestep γ is executed. Let $I'_\gamma = \{t_\gamma, \ldots, t\}$, and let R_t be the random variable denoting the total number of rounds of steal attempts in $\mathcal{T}_\gamma \cup I'_\gamma$.

Proposition 4.2 implies that there exists some constant $c' > 0$ such that $\mathbf{Pr}[R_t \leq c'\beta_\varepsilon] \geq 1 - \varepsilon/2$. Then,

$$
\begin{aligned}
&\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon] \\
\geq\;& \mathbf{Pr}[(\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon) \cap (R_t \leq c'\beta_\varepsilon)] \\
=\;& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon \mid R_t \leq c'\beta_\varepsilon]\,\mathbf{Pr}[R_t \leq c'\beta_\varepsilon] \\
\geq\;& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon \mid R_t \leq c'\beta_\varepsilon]\,(1 - \varepsilon/2)\,.
\end{aligned}
$$

If $R_t \leq c'\beta_\varepsilon$, then Proposition 4.3 implies that H_t, the number of heavy premature nodes at t, is at most $c_1 P\beta_\varepsilon$, $c_1 > 0$. Then, $\sum_{i=1}^{H_t} K_{t_i} \leq \sum_{i=1}^{c_1 P\beta_\varepsilon} K_{t_i}$, where we set $K_{t_i} = K_{t_1}$, for all $i > H_t$. It follows that:

$$
\begin{aligned}
&\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\beta_\varepsilon \lambda_\varepsilon] \\
\geq\;& \mathbf{Pr}[\sum_{i=1}^{c_1 P\beta_\varepsilon} K_{t_i} \leq (c_1 P\beta_\varepsilon) c_2 \lambda_\varepsilon \mid R_t \leq c'\beta_\varepsilon]\,(1 - \varepsilon/2) \\
\geq\;& \mathbf{Pr}[(\bigcap_{i=1}^{c_1 P\beta_\varepsilon} (K_{t_i} \leq c_2 \lambda_\varepsilon) \mid R_t \leq c'\beta_\varepsilon]\,(1 - \varepsilon/2)
\end{aligned}
$$

$$\geq \quad (1 - \sum_{i=1}^{c_1 P \beta_\varepsilon} \mathbf{Pr}[K_{t_i} > c_2\, \lambda_\varepsilon \mid R_t \leq c'\, \beta_\varepsilon])\, (1 - \varepsilon/2)\,.$$

For any $i \in [c_1 P \beta_\varepsilon]$, it can be proved that $\mathbf{Pr}[K_{t_i} > c_2\, \lambda_\varepsilon \mid R_t \leq c'\, \beta_\varepsilon] \;<\; \varepsilon/(2c_1 P \beta_\varepsilon)$. Thus,

$$\begin{aligned}\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1\, c_2\, \beta_\varepsilon\, \lambda_\varepsilon] \quad &\geq \quad (1 - \frac{c_1\, P\, \beta_\varepsilon\, \varepsilon}{2c_1\, P\, \beta_\varepsilon})(1 - \varepsilon/2) \\ &= \quad (1 - \varepsilon/2)\,(1 - \varepsilon/2) \\ &\geq \quad 1 - \varepsilon\,,\end{aligned}$$

as needed. ■

4.3 Time Bound

To bound the time complexity of our algorithm we first bound the time spent to execute a scheduling iteration. It turns out that the time spent for each scheduling iteration i is $O((E_i/P + 1)\log(|\mathcal{B}_i| + P))$, where E_i is the random variable denoting the number of threads that are enabled in this iteration. By using conditional expectations, it can be proved that, for any random variable N with $\mathcal{E}(N) < \infty$, (1) $\mathcal{E}(\sum_{i=1}^{N}(E_i \log(|\mathcal{B}_i| + P)/P)) \in O(\sigma \log(PT_\infty)/P)$, and (2) $\mathcal{E}(\sum_{i=1}^{N} \log(|\mathcal{B}_i| + P)) \in O(\mathcal{E}(N)\log(PT_\infty))$. (Recall that σ is the total number of synchronization edges of the computation.) We use these facts to prove our time bound. More specifically, for the time complexity of our algorithm we prove:

Theorem 4.6 *For any computation G and any integer $P \in \Omega(\log_P \log T_\infty)$, it holds that $\mathcal{E}(T_{\mathtt{DFPS}}(G, P)) \in O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$.*

Sketch of proof: We partition the execution of our algorithm into epochs, where each epoch i consists of the timesteps comprising the ith scheduling iteration. We then classify all epochs into *stealing-concentrated* epochs, that is, epochs during which a lot of steal attempts are performed, and *execution-concentrated* epochs, that is, epochs during which less stealing is performed and thus a lot of instructions are executed. We further classify stealing-concentrated epochs into those where less than P' ready lists are in the working tree when the epoch starts, and those where at least P' ready lists are in the working tree when the epoch starts. We use the fact that T_1 is an upper bound on the number of the executed nodes and we employ several different techniques to bound the number of timesteps in each of these types of epochs. We use useful properties of the lists of threads (some of which have been already presented in Section 4.1) to bound the expected number of timesteps in stealing-concentrated epochs with less than P ready lists in the working tree. The most difficult type of epochs appears to be the case of stealing-concentrated epochs with at least P' ready lists. In this case, we derive an interesting relation to an occupancy problem in which $P \log P$ balls are placed uniformly at random into P, initially empty, bins (see [23, Section 4.4.4]). We prove that the probability that the number of occupied bins is $O(P)$ is at least $1 - 2e^{(P-1)/8}$. We define a successful such epoch in such a way that the probability of success of an epoch is the same as the probability that the number of occupied bins is $O(P)$ in the above occupancy problem. We use this fact to bound the expected number of timesteps in both successful and unsuccessful epochs of this type. ■

4.4 Contention

In our time analysis, we didn't account for contention caused by competing processors. The following theorem says that the processors do not cause any serious contention for accessing the working arrays.

Theorem 4.7 *Consider any multithreaded computation G, let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathtt{DFPS}}(G, P)$ is any execution schedule produced by algorithm* DFPS *and let e be some arbitrary position of any of the arrays Σ, Π, Δ and E. Then, at most 3 processors may contend for accessing e.*

Notice that under a non-blocking implementation of the ready lists (e.g., the one presented in [1]), processors do not wait in queues when they try to steal from the same ready list. We can thus conclude that accessing any of the working arrays do not cause any serious contention (during the execution of DFPS). (In previously presented depth-first scheduling algorithms [4, 5, 6, 25], the working processors cause a serious amount of contention because they all try to access the same end-points of some arrays.)

4.5 Large Allocations

In our space analysis, we assumed that every instruction γ allocates at most K units of memory, where K is the memory threshold at the time instruction γ is executed. Our algorithm handles large allocations as described in Section 3. Before every allocation of $m > K$ bytes, it inserts a binary fork tree of depth $\log(m/\log P)$ so that $m/\log P$ dummy threads are created at its leaves. (Notice that for our algorithm it always holds that $K > \log P$, so that $m/K < m/\log P$). Clearly, each one of the $m/\log P$ dummy threads is allowed to allocate K space, which it does not actually allocate. Thus, executing the large-allocation instruction after the execution of these dummy threads does not violate our space quota concept. As in [24], we assume that an allocation of M bytes of memory has depth $O(\log M)$. This assumption is reasonable in systems with binary forks that clear the memory as soon as it is allocated; this clearing is usually performed in parallel by forking a tree of height $\Theta(\log M)$.

Let S_K be the *excess allocation* in the parallel computation, defined as the sum of all memory allocations greater than K (for any particular value of K during the execution). In our cost model the creation of dummy threads increases the depth T_∞ of our computation by only a constant factor. The work of the computation, as well as the number of synchronizations are increased by $S_K/\log P$. Thus, our space bound remains unchanged, while our time bound becomes $\mathcal{E}(T_{\mathtt{DFPS}}(G, P)) \in O(T_1/P + (\sigma + S_K)\log(PT_\infty)/P + T_\infty \log(PT_\infty))$.

5 Acknowledgments

We would like to thank Anil Kumar and Paul Spirakis for several useful discussions. Many thanks to Faith Fich for her invaluable feedback in improving the presentation of this paper. Our thanks go to the anonymous SPAA'01 reviewers for their helpful comments.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on*

Parallel Algorithms and Architectures, pages 119–129, Puerto Vallarta, Mexico, June-July 1998.

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

[3] G. E. Blelloch, S. Chattejee, 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. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12. Santa Barbara, California, July 1995.

[5] G. E. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, March 1999.

[6] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23. Newport, Rhode Island, June 1997.

[7] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, September 1995.

[8] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

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

[11] R. Chandra, A. Gupta, and J. L. Hennessy. Cool: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[12] K. Chandy and C. Kesselman. Compositional c++: compositional parallel programming. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 124–144. New Haven, August 1992.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[14] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150. Honolulu, Hawai, May-June 1988.

[15] P. Fatourou and P. Spirakis. A new scheduling algorithm for general strict multithreaded computations. In *Proceedings of the 13rd International Symposium on DIStributed Computing*, pages 297–311. Bratislava, Slovakia, September 1999.

[16] P. Fatourou and P. Spirakis. Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems Journal*, 33(3):173–232, May-June 2000.

[17] W. Feller. *An introduction to probability theory and its applications, vol. 1*. 3rd edition, Wiley, 1971.

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

[19] 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.

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

[21] L. Kale, B. Ramkumar, V. Saletore, and A. Sinha. Prioritization in parallel symbolic computing. In *Parallel Symbolic Computing: Languages, Systems, and Applications*. Cambridge, USA, October 1992.

[22] D. Kranz, R. Halstead, and E. Mohr. Mul-t: A high-performance parallel lisp. In *Proceedings of the Conference on Programming Language Design and Implementation*. Portland, Oregon, June 1989.

[23] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[24] G. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 83–95. Saint-Malo, France, June 1999.

[25] G. Narlikar. *Space-Efficient Scheduling for Parallel, Multithreaded Computations*. PhD thesis, School of Computer Science, Carniege Mellon University, May 1999.

[26] G. Narlikar and G. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[27] G. Narlikar and G. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of SuperComputing '98: High Performance Networking and Computing*, November 1998.

[28] M. Rinard, D. Scales, and M. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.