I N F O R M A T I K

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

Panagiota Fatourou

MPI–I–2000–1–003 May 2000

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Im Stadtwald 66123 Saarbrücken Germany

Authors' Addresses

Panagiota Fatourou
Max-Planck-Institut für Informatik
D-66123 Saarbrücken
Germany
faturu@mpi-sb.mpg.de

Abstract

In this paper, we present a randomized, online, space-efficient algorithm for the general class of programs with synchronization variables (such programs are produced by parallel programming languages, like, e.g., Cool, ID, Sisal, Mul-T, OLDEN and Jade). The algorithm achieves good locality and low scheduling overheads for this general class of computations, by combining *work-stealing* and *depth-first* scheduling.

More specifically, given a computation with work T_1, depth T_∞ and σ synchronizations that its execution requires space S_1 on a single-processor computer, our algorithm achieves expected space complexity at most $S_1 + O(PT_\infty \log(PT_\infty))$ and runs in an expected number of $O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$ timesteps on a shared-memory, parallel machine with P processors. Moreover, for any $\varepsilon > 0$, the space complexity of our algorithm is at most $S_1 + O(P(T_\infty + \ln(1/\varepsilon)) \log(P(T_\infty + \ln(P(T_\infty + \ln(1/\varepsilon))/\varepsilon))))$ with probability at least $1 - \varepsilon$. 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 $1 - e^{-T_\infty}$. The algorithm achieves good locality and low scheduling overheads by automatically increasing the granularity of the work scheduled on each processor.

Our results combine and extend previous algorithms and analysis techniques (published by Blelloch *et. al* in SPAA'97 [6] and by Narlikar in SPAA'99 [26]). Our algorithm not only exhibits the same good space complexity for the general class of programs with synchronization variables as its deterministic analog presented in [6], but it also achieves good locality and low scheduling overhead as the algorithm presented in [26], which however performs well only for the more restricted class of *nested parallel* computations.

Keywords

Multithreaded computations, depth-first scheduling, work-stealing scheduling, randomized scheduling algorithms, online schedulers, parallel programming languages, synchronization variables.

1 Introduction

Multithreaded models of parallel computation have typically been proposed as a general approach to model dynamic, unstructured parallelism, since many parallel programming languages allow the expression of dynamic threads. Such languages include data-parallel languages like High Performance Fortran [19] and NESL [4], dataflow languages like ID [13], control-parallel languages with fork-join constructs like CILK [7], CC++ [11] and Proteus [23], languages with synchronization primitives like Mul-T [22], COOL [10], Jade [29], SISAL [16], and various other user-level thread libraries.

For the execution of a multithreaded computation on a parallel computer, one should specify which processor executes which threads and when each thread should be executed. A good scheduler must ensure that enough threads remain active to keep the processors busy, which guarantees good running time. On the other hand, the scheduler must keep the concurrently active threads into limits so as to guarantee good space complexity. Since parallel programs solve large problems it is not rarely the case that their performance degrades due to limitations in the available memory. Thus, another important objective of an efficient scheduling algorithm is to obtain good locality, because the local caches of today's hardware-coherent shared memory multiprocessors are large and improving their hit ratio significantly improves performance. Good locality is usually obtained by scheduling threads close in the computation graph on the same processor. These threads usually share common data. Additionally, the scheduler must incur low overheads and must provide fast operations like creation, deletion and synchronization of threads. Apparently, designing a scheduling algorithm to achieve all of the above goals is not a trivial task.

Several systems have used work-stealing to achieve the scheduling goals described above. *Work-stealing* is a technique in which underutilized processors try to "steal" work from other, hopefully over-utilized processors. Blumofe and Leiserson [9, 7] were the first to prove that work-stealing algorithms are efficient in terms both of space complexity and execution time. Their randomized scheduling algorithm [9] requires $O(T_1/P + T_\infty)$ (expected) time and $O(PS_1)$ space to execute a *fully-strict* multithreaded computation with work T_1, depth T_∞ and serial, depth-first space S_1 on a P-processor parallel computer. Fatourou *et al.* has presented in [14, 15] provably efficient scheduling algorithms for more general classes than the fully-strict class of multithreaded computations. Their results [14, 15] generalize Blumofe and Leiserson's results [9, 7] for the class of programs that follow the general strict paradigm of parallelism.

Recently, a flurry of research has resulted in *depth-first* schedulers, that is schedulers that schedule threads prioritized by their serial depth-first execution order. These schedulers achieve space complexity $S_1 + O(PT_\infty)$, which for sufficient parallelism is just an additive factor over the sequential space. Thus, the space bound achieved by depth-first schedulers improves upon the previous bound of $O(PS_1)$ for computations with sufficient parallelism[1]. Moreover, depth-first schedulers cope with a more general memory allocation model than the stacked-based allocations assumed in previous works on work-stealing (see e.g., [8]). However, depth-first schedulers use a globally ordered centralized data structure of batches of

[1]Intuitively, space complexity $O(PS_1)$ means that, on the average, each of the P processors needs no more memory than S_1, the space used by the optimal one-processor execution. For languages with sufficient parallelism (that is, when T_∞ is small), the space complexity of depth-first schedulers is close to the optimal space bound.

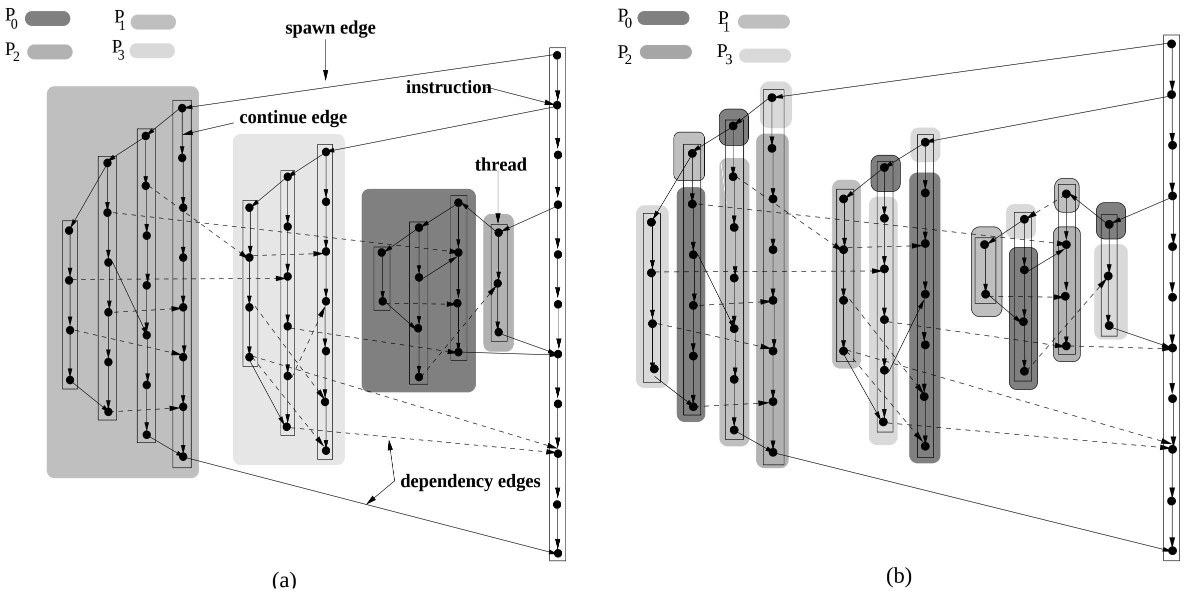

Figure 1: Scheduling granularities of two different algorithms. Circles represent serially-executed, unit-time instructions, while the boxes of vertical chains of circles represent threads. The vertical edges, called *continue edges*, determine the execution ordering of the instructions of the threads, the right-to-left edges, called *spawn edges*, represent spawning of new threads which can be executed in parallel to the parent thread, while accesses of different threads to synchronization variables is represented by left-to-right edges, called *dependency edges*. The rounded boxes represent how threads are mapped onto processors. Clearly, an algorithm that achieves the first mapping is significantly better in terms of locality and contention than some algorithm that results in the second mapping.

ready threads and thus they do not enjoy some of the practical advantages provided by work-stealing schedulers [9, 7, 14, 15]. When the threads are fine-grained, the performance of the algorithms may suffer due to poor locality and high scheduling contention for accessing shared data structures. Narlikar [25] 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 typically implies better locality and lower scheduling contention. Apparently, depth-first scheduling algorithms are efficient when threads are coarsened and their granularity is large. Figure 1 represents two different mappings of the threads of some computation onto processors. Each rounded box declares the batch of threads/instructions scheduled together on the same processor. Clearly, scheduling threads close together in the graph on the same processor, as in Figure 1(a), provides better cache locality, and thus lower contention.

The first depth-first scheduling algorithm was presented by Blelloch *et. al* in [5] for languages with *nested fine-grained* parallelism. Their algorithm works in such a way that threads need to be rescheduled after every unit computation to guarantee their good space bound, and thus, the algorithm ignores the issue of locality and results in high contention. More recently, Narlikar and Blelloch [28] showed that the same good bounds presented in [5] can be achieved for an algorithm that moves threads from a processor when they fork, synchronize or allocate large amounts of memory. However, even this algorithm can result in poor locality, since although a single thread may execute on the same processor, threads close together in the computation graph are usually scheduled on different processors. It was very recently that Narlikar in [25] has presented a scheduling algorithm, called DFDeques, that allows to processors to work on different (globally ordered) queues of ready threads. This results in better locality and in lower contention during scheduling.

However, all the above presented work on depth-first scheduling has been limited in computations which are nested parallel (or fully-strict), that is computations in which threads are allowed to synchronize only with their parent thread. This class of programs appears to be quite restricted, since it does not include for example languages based on futures [10, 18, 22], languages based on lenient or speculative evaluation [2] or languages with general user-specified synchronization constraints [29]. The only work that extends the depth-first approach to a more general class of computations is by Blelloch *et al.* [6]. The algorithm presented in [6] works for computations with *write-once* synchronization variables, that is, variables that can be written by one thread and read by any number of other threads. When an instruction reads a synchronization variable, a *dependency edge* exists in the computation (see Figure 1(a)) from the instruction that writes the synchronization variable to the instruction that reads it. A *suspended* thread is one that waits for a synchronization variable to be written. After the variable is written, all suspended threads on the variable are *enabled*. Blelloch *et. al* discuss in [6, Section 2] how the write-once synchronization variables can be used to implement I-structures [2], futures [10, 18, 22], streams and are likely to be helpful in implementing the user-specified synchronization constraints in Jade [29]. Their algorithm, which is deterministic, exhibits space complexity $S_1 + O(PT_\infty \log(PT_\infty))$ and runs in $O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$ time, where σ is the number of synchronizations in the computation graph. The major disadvantages of their algorithm are the same as those for all works on depth-first schedulers before Narlikar's work [25]. Thus, the scheduling granularity of the algorithm is small and the algorithm presents high contention and not good locality. For instance, their algorithm although may schedule one thread on the same processor, typically schedules close together threads on different processors, while threads are moved to the centralized data structure every time they fork, suspend, enable some other thread or allocate large amounts of memory.

In this work, we present a scheduling algorithm that enjoys the good characteristics of low contention and good locality of algorithm DFDeques presented by Narlirkar in [25] but for the general class of computations with write-once synchronization variables studied by Blelloch *et al.* in [6]. Our algorithm not only exhibits the good space-complexity of the depth-first schedulers, but also presents good scheduling granularity and thus exhibits lower contention than the algorithm presented in [6]. Our algorithm is randomized and employs the work-stealing paradigm, combined with the depth-first property in order to keep its memory requirements low.

More specifically, our algorithm exhibits expected space complexity at most $S_1 + O(PT_\infty \log(PT_\infty))$, and expected running time $O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$. Moreover, for any $\varepsilon > 0$, our algorithm requires at most $S_1 + O(P(T_\infty + \ln(1/\varepsilon)) \log(P(T_\infty + \ln(P(T_\infty + \ln(1/\varepsilon))/\varepsilon))))$ space, with probability at least $1-\varepsilon$. It is not difficult to verify that even for values of ε as small as e^{-T_∞}, it holds that $O(P(T_\infty + \ln(1/\varepsilon)) \log(P(T_\infty + \ln(P(T_\infty + \ln(1/\varepsilon))/\varepsilon)))) = O(PT_\infty \log(PT_\infty))$. These bounds include all time and space costs for both the computation and the scheduler.

Our algorithm uses the synchronization queues, described in [17] and used in [6], in order to maintain efficiently the sets of suspended and enabled threads. The algorithm employs the black-white priority tree presented in [6]. However, instead of storing threads at the leaves (as in [6]) it stores lists of threads, which are globally ordered according to their serial execution order. The algorithm borrows ideas from Narlikar's algorithm [25] for reducing contention and improving scheduling granularity. More specifically, 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 ready list chosen uniformly at random from a set of high priority lists of ready threads. The algorithm presents good scheduling granularity since it schedules on the same processor threads belonging to the same ready list. Moreover, stolen threads are topmost in their ready lists, that is threads which are expected to be the coarsest threads in the lists [9, 7] (this claim has also been proved experimentally in [25]).

The processors executing the computation run asynchronously and they content only when they try to steal from the same ready list as other processors. On the contrary, in the algorithm presented in [6], these processors were all trying to obtain work from the same end-point of an array containing ready threads, while when threads were preempted, the processors were trying to access the same end-point of some other array in order to write data about the state of these threads. In our algorithm processors choose uniformly at random positions in the array that they need to access and they content only with at most two more processors when they store preempted lists of threads. Moreover, the granularity achieved by our scheduler is larger than the one of the algorithm in [6]. Thus, our algorithm is significantly better than the one in [6] in terms of good locality and low overheads due to contention. For example, our algorithm is likely to achieve the mapping provided in Figure 1(a), while it is usually the case that the algorithm in [6] results in the mapping presented in Figure 1(b).

Our algorithm extends Narlikar's algorithm [25] to work for more general computations, those with synchronization variables. Scheduling the more general class of computations is more complicated than for nested-parallel computations considered in past, since dependency edges are no longer "localized" (a thread may synchronize with an arbitrary 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 program.

Our analysis extends several parts of the analysis presented in [25] and proves that the space bound analysis presented in [25] can be applied even in more general cases if specific significant properties, figured out in our analysis, hold. Our space bound analysis is though much more complicated than the one presented in [25], because our algorithm employs variable *memory quota* during its execution. Additionally, proving the time bound of our analysis not only combines ideas from [6] and [25] but also employs new interesting techniques. Narlikar's bounds [25] (under a simplistic model not counting for contention) are slightly increased (by a multiplicative factor of $\log P$) than previous bounds of space-efficient schedulers for *nested-parallel* computations (under the same model). Our bound on the other hand, do not diverge from the previous bound presented in [6] for computations with write-once synchronization variables, although our algorithm enjoys similar nice properties over Blelloch's *et. al* work [6] as Narlikar's algorithm [25] over previous works [5, 28].

Since our algorithm is a depth-first algorithm, one of its major objectives is to be space-efficient. Apparently, this objective has been achieved by our algorithm. Time-efficient algorithms, which however ignore the space requirements, have been presented in the literature (see, e.g., [1]) even for general computations.

This paper is organized as follows. Section 2 describes some mathematical machinery. Section 3 includes definitions and some preliminary facts, while Section 4 presents our algorithm. Section 5 exhibits fundamental properties maintained by the algorithm, which are important for proving later in the same section our space and time complexity bounds, while Section 6 considers the case of large allocations. We conclude in Section 7, with a discussion of our results and some open problems.

2 Mathematical Foundations

For each integer $n \geq 1$, denote $[n] = \{1, \ldots, n\}$ and let $\log n$ stand for $\log_2 n$. For each integer $n > 0$, all arrays of n elements are indexed from 1 to n. This indexing scheme is generalized for multidimensional arrays. For any function ϕ, we denote ϕ'' the second derivative of ϕ.

Let X be a random variable and consider any value x that it assumes. We denote by $\mathbf{Pr}[X = x]$ the probability of the event $X = x$, while $\mathcal{E}(X)$ stands for the *expectation* of X. Recall that expectation distributes over any linear combination of random variables.

2.1 The Conditional Expectation

In this section we describe conditional expectations. Our presentation closely follows the one presented in [24, Section 4.4, Chapter 4].

Let X, Y be two random variables. The random variable $\mathcal{E}(X \mid Y)$ is defined to be the random variable $f(y)$ such that $f(y) = \mathcal{E}(X \mid Y = y)$. Roughly speaking, $\mathcal{E}(X \mid Y = y)$ is the conditional expectation of X given that Y equals y. The above definition of the conditional expectation is generalized to define the random variable $\mathcal{E}(X \mid Y_1, \ldots, Y_r)$, where $r > 0$ is any integer, in the natural way.

For the conditional expectation we know (see e.g., [24, Lemma 4.9, Chapter 4]) the following:

Lemma 2.1 *It holds that:*

$$\mathcal{E}(\mathcal{E}(X \mid Y)) \quad = \quad \mathcal{E}(X)\,.$$

The above lemma can be easily generalized to random variables of the form $\mathcal{E}(X \mid Y_1, \ldots, Y_r)$ (see e.g., [24, Lemma 4.9, Chapter 4]).

2.2 The Chernoff Bounds

In this section, we describe the well-known Chernoff bounds [24, Exercise 4.1 and Theorem 4.2, Chapter 4], classical results from probability theory.

Let $X_1, \ldots, X_n$ be independent Bernoulli trials and let $X = \sum_{i=1}^{n} X_i$. Lemma 2.2 [24, Exercise 4.1, Chapter 4] considers deviations of X above its expectation.

Lemma 2.2 *Let $X_1, \ldots, X_n$ be independent Bernoulli trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, it holds that $\mu = \mathcal{E}(X) = \sum_{i=1}^{n} p_i$ and for any $\delta > 0$,*

$$\mathbf{Pr}[X > (1 + \delta)\mu] \quad < \quad \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}\,.$$

Lemma 2.3 [24, Theorem 4.2, Chapter 4] considers deviations of X below its expectation.

Lemma 2.3 *Let $X_1, \ldots, X_n$ be independent Bernoulli trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, it holds that $\mu = \mathcal{E}(X) = \sum_{i=1}^{n} p_i$ and for any $\delta > 0$,*

$$\mathbf{Pr}[X < (1-\delta)\mu] \quad < \quad \exp\left\{-\frac{\mu\delta^2}{2}\right\}.$$

2.3 An Occupancy Problem

In this section, we describe an interesting occupancy problem and we present useful properties of random variables related to the problem.

We have m balls and we want to place them independently and uniformly into n bins. Let Z denote the number of bins that remain empty. Then, the following lemma holds (see e.g., [24, Theorem 4.18, Section 4.4.4]).

Lemma 2.4 *Let $m, n > 0$ be any two integers. Assume that $r = m/n$, and let Z be the number of empty bins when m balls are thrown randomly into n bins. Then,*

(1) $\mathcal{E}(Z) = n\left(1 - \frac{1}{n}\right)^m \sim ne^{-r}$, *and*

(2) *for any* $\delta > 0$,

$$\mathbf{Pr}[|Z - \mathcal{E}(Z)| \geq \delta] \quad \leq \quad 2\mathrm{exp}\left(-\frac{\delta^2(n-1/2)}{n^2 - [\mathcal{E}(Z)]^2}\right).$$

Let X be the random variable representing the number of bins that are occupied when the number of balls in the above described occupancy problem is $m = P\log P$ and the number of bins is $n = P$. Then, by using Lemma 2.4, it can be proved:

Lemma 2.5 *Let $P > 2$ be any integer and X be the random variable denoting the number of bins that are occupied when $P\log P$ balls are placed independently and uniformly at random into P bins. Then,*

(1) $\mathcal{E}(X) \geq P - 1$, *and*

(2) $\mathbf{Pr}[X > (P-1)/2] \geq 1 - 2e^{-(P-1)/8}$.

Proof: We start by proving (1). Let Z be the random variable denoting the number of empty bins. Then, $X = P - Z$, so that, by linearity of expectation, $\mathcal{E}(X) = P - \mathcal{E}(Z)$. Lemma 2.4 implies that

$$\begin{aligned}
\mathcal{E}(Z) &\sim Pexp\left\{-\frac{P\log P}{P}\right\} \\
&= \frac{P}{e^{\log P}} \\
&\leq \frac{P}{e^{\ln P}} \\
&= 1\,.
\end{aligned}$$

Thus, $\mathcal{E}(X) = P - \mathcal{E}(Z) \geq P - 1$, as needed.

We continue to prove (2). Lemma 2.4 implies that for any $\delta > 0$

$$\mathbf{Pr}[|Z - \mathcal{E}(Z)| \geq \delta] \quad \leq \quad 2\exp\left\{-\frac{\delta^2(P-1/2)}{P^2 - [\mathcal{E}(Z)]^2}\right\}.$$

Clearly, $\mathbf{Pr}[Z \geq \mathcal{E}(Z) + \delta] \leq \mathbf{Pr}[|Z - \mathcal{E}(Z)| \geq \delta]$, so that

$$\begin{aligned}\mathbf{Pr}[Z \geq \mathcal{E}(Z) + \delta] \quad &\leq \quad 2\exp\left\{-\frac{\delta^2(P-1/2)}{P^2 - [\mathcal{E}(Z)]^2}\right\} \\ &\leq \quad 2\exp\left\{-\frac{\delta^2(P-1/2)}{P^2}\right\}.\end{aligned}$$

Since $\mathcal{E}(Z) \leq 1$, $\mathbf{Pr}[Z \geq 1 + \delta] \leq \mathbf{Pr}[Z \geq \mathcal{E}(Z) + \delta]$. Since $Z = P - X$, $\mathbf{Pr}[Z \geq 1 + \delta] = \mathbf{Pr}[P - X \geq 1 + \delta] = \mathbf{Pr}[X \leq P - 1 - \delta]$, so that

$$\mathbf{Pr}[X \leq P - 1 - \delta] \quad \leq \quad 2\exp\left\{-\frac{\delta^2(P-1/2)}{P^2}\right\}.$$

Let $\delta = (P-1)/2 > 0$. Then, $\mathbf{Pr}[X \leq P - 1 - \delta] = \mathbf{Pr}[X \leq (P-1)/2]$. Thus,

$$\begin{aligned}\mathbf{Pr}[X \leq (P-1)/2] \quad &\leq \quad 2\exp\left\{-\frac{(P-1)^2\,(P-1/2)}{4P^2}\right\} \\ &\leq \quad 2\,e^{-(P-1)/8},\end{aligned}$$

whenever $P > 2$. Clearly, $\mathbf{Pr}[X > (P-1)/2] = 1 - \mathbf{Pr}[X \leq (P-1)/2]$, so that

$$\mathbf{Pr}[X > (P-1)/2] \quad \geq \quad 1 - 2\,e^{-(P-1)/8}.$$

as needed. ■

2.4 The Synchronization Queue

In this section, we describe a data structure, called synchronization queue, which was first introduced and analyzed in [17, Section 4.1]. Our presentation closely follows the one in [17].

A *synchronization queue* [17, Section 4.1] is a queue supporting the following three operations: (1) create an empty queue, (2) enqueue elements, and (3) dequeue all elements. A synchronization queue q is implemented[2] as a pair of an array q_a and an integer q_l, where q_l is the length of the queue, and $q_l \leq |q_a|$.

For a synchronization queue the following proposition has been proved in [17].

Proposition 2.6 (Greiner & Blelloch) *Using synchronization queues, each of the three synchronization queue operations on m elements can be performed in $O(m/P)$ time on a P-processor CRCW PRAM.*

[2]More details on the implementation of the synchronization queues are provided in [17].

3 Definitions & Preliminaries

In this section, we present formal definitions and some preliminary facts. Our definitions closely follows the ones presented in [15, Section 2], [26, Chapters 2 & 6] and [6, Section 2].

The rest of this section is organized as follows. The basic entity of a multithreaded computation is introduced in Section 3.1. The concepts of ready nodes and execution schedules are described in Section 3.2. We treat scheduling algorithms in Section 3.3, while Section 3.4 describes useful tools for the analysis of our algorithm.

3.1 Multithreaded Computations

A computation is modeled as a set of threads, each comprised of a sequence of unit-time, serially-executed instructions. Each thread can *fork* (or *spawn*) other threads. Threads can synchronize through the use of synchronization variables. All threads share a single address space. As with the work of Blelloch *et. al* [6] we assume each thread executes a standard RAM instruction set augmented with instructions for (1) creating a new thread to run in parallel with its parent, (2) allocating and releasing a block of memory, (3) reading and writing to a regular memory location, and (3) allocating, reading, writing and releasing a synchronization variable. Each synchronization variable can only be written once, but it can be read by any number of threads. A thread that performs a read on an unwritten variable *stalls* (or *suspends*). The thread is *enabled* when the variable is written by some other thread.

We associate a dynamic directed acyclic graph (or dag) G to any computation of this model. Each node in the dag represents a single unit-time instruction, and the edges represent ordering constraints. There are three kind of edges in the dag: continue edges, spawn edges, and dependency edges. The nodes of a thread are linked by *continue edges*, forming a chain corresponding to the dynamic instruction execution order of the thread. During the course of a thread's execution, the thread may create, or *spawn* child threads, which can be executed in parallel with the parent thread. If an instruction γ of a thread Γ spawns another thread Γ', a *spawn edge* begins from node γ and ends at the first node of thread Γ' in the dag. We assume that a node can spawn only one thread. Whenever threads synchronize, such that an instruction in one thread cannot be executed until after some instruction in another thread, then the dag contains an edge from the node representing the latter instruction to the node representing the former instruction. We say that the thread containing the former instruction *stalls* (or suspends) waiting for the latter instruction to be executed. Once the latter instruction is executed, the thread containing the former instruction is *enabled* and its execution can proceed. In our model, when an instruction reads a synchronization variable, a dependency edge exists in the dag from the instruction that writes the synchronization variable to the instruction that reads it. If the read instruction is executed first the thread containing the instruction suspends, waiting for the write instruction to be executed.

A *synchronization edge* is a spawn or a dependency edge. For any multithreaded computation G, we denote by $E_\sigma(G)$ the set of synchronization edges of G and we let $\sigma = |E_\sigma(G)|$; that is, σ is the total number of spawn and dependency edges in the dag. A thread *terminates* when all its instructions have been executed; a *dead* thread is one that has terminated. A spawned thread that is not dead is *alive*. For every edge $\langle\gamma, \gamma'\rangle$ in the DAG, 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

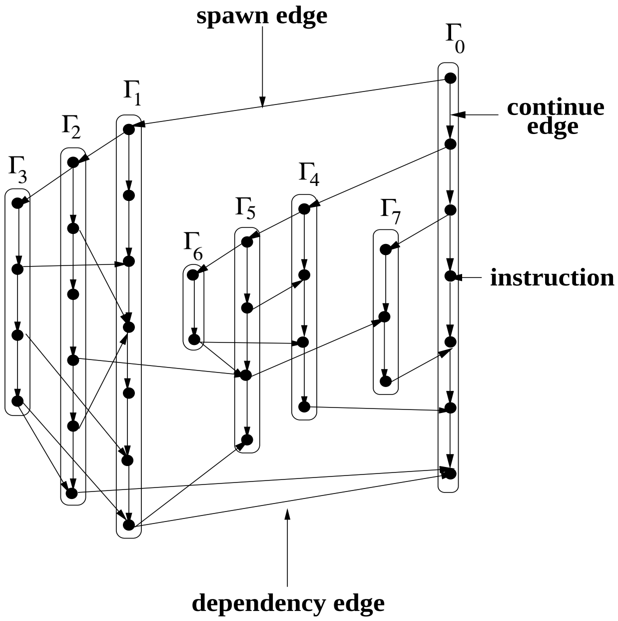

Figure 2: A multithreaded computation. A thread is represented by a block of circles, where each circle represents a particular instruction of the thread. The vertical edges are continue edges. The right-to-left edges are spawn edges, while the left-to-right edges are dependency edges.

executed in its thread.

Figure 2 represents a multithreaded computation. A thread in this graph is represented by a block of circles, where each circle represents a particular instruction of the thread. The vertical edges in Figure 2 define the serial order in which the instructions of a thread are executed. The right-to-left edges represent a fork of a new thread, while each left-to-right edge represents a synchronization between two threads.

Each instruction node γ in the dag is assigned an integer weight $\mu(\gamma)$ to represent a memory allocation or deallocation. Thus, a node γ that allocates M units of memory is assigned a weight $\mu(\gamma) = M$, while if the node deallocates M bytes, then $\mu(\gamma) = -M$. If the node performs no allocation or deallocation, $\mu(\gamma) = 0$. If γ allocates a synchronization variable, $\mu(\gamma) = 1$.

The *work* of a computation is the total number of nodes of the dag. The *dag depth of a node* in G is the length of the longest path of the instruction graph that terminates at the node. The *dag depth of* G is the maximum dag depth of any of its nodes. We denote by $T_1(G)$ the minimum execution time for executing G serially. Notice that $T_1(G)$ is the work of the computation, since a one-processor computer can only execute one instruction at any timestep. We denote by $T_\infty(G)$ the dag depth of G, since even with arbitrarily many processors each instruction on a path must execute serially. For the computation of Figure 2, it can be easily derived that $T_\infty(G) = 12$, while $T_1(G) = 37$.

3.2 Ready Nodes - Execution Schedules

At any given timestep t during the execution of a multithreaded computation G, an unexecuted node is *ready* if all its predecessors in the instruction dag have been executed. Notice that a thread can not contain more than one ready node. A thread that contains a ready node is called *ready*. The ready node of a thread is called the thread's *current* node. Alive threads that are not ready are considered suspended.

Roughly speaking, an execution schedule for a multithreaded computation G determines which processors of the parallel computer execute which instructions at each step. In any given step of an execution schedule, each processor either executes a single instruction or sits idle. Let t be any timestep during the execution of G. More formally, a *P-processor configuration at timestep t* is a tuple $\mathcal{X}_t(G,P)$ of P elements $\langle\gamma_1,\gamma_2,\ldots,\gamma_P\rangle$ such that for each $i \in [P]$, either $\gamma_i = \gamma$ if processor i executes the *ready* node γ at timestep t, or $\gamma_i = \emptyset$ if processor i does not execute any instruction at timestep t; roughly speaking, for every processor p, $\mathcal{X}_t(G,P)$ determines which ready node (if any) is executed by this processor at timestep t. A *P-processor execution schedule for a multithreaded computation G* is a sequence $\mathcal{X}(G,P) = \mathcal{X}_1(G,P), \mathcal{X}_2(G,P), \ldots \mathcal{X}_T(G,P)$, such that all nodes of the dag are contained in one of the $\mathcal{X}_i(G,P)$, $i \in [T]$; we say that $\mathcal{X}(G,P)$ has length T. Thus, the length of any execution schedule is the time required to execute it, that is, the number of timesteps in it. We denote by $|\mathcal{X}(G,P)|$, the length of an execution schedule $\mathcal{X}(G,P)$. The space required by $\mathcal{X}(G,P)$, denoted $S(\mathcal{X}(G,P))$, is defined to be $S(\mathcal{X}(G,P)) = n + \max_{t=1,\ldots,|\mathcal{X}(G,P)|}\{\sum_{i=1}^{t}\sum_{\gamma\in\mathcal{X}_i(G,P)\&\gamma\neq\emptyset}\mu(\gamma)\}$, where n is the input data.

The *DF-schedule* [5, 6, 25] for a multithreaded computation G is defined by introducing a *serial priority order* on the live threads; that is, we introduce a total order on the live threads such that $\Gamma_\alpha \succ \Gamma_\beta$ means that thread Γ_α has a higher priority than thread Γ_β. The rule for deriving this ordering is the following. Whenever a thread spawns another thread, the spawned thread will have a larger priority than the spawning thread but the same priority relative to all other live threads. The *DF-schedule*, denoted $\mathcal{X}_{DF}(G)$ is the one-processor 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. Clearly, at each timestep of the $\mathcal{X}_{DF}(G)$ exactly one node is executed. Assume that $\langle\gamma_1,\ldots,\gamma_{|\mathcal{X}_{DF}(G)|}\rangle$ are the nodes of a computation G, in the order in which they are executed in $\mathcal{X}_{DF}(G)$. The *serial space requirement* [5, 6, 25], denoted by S_1, is defined to be $S_1 = n + \max_{t=1,\ldots|\mathcal{X}_{DF}(G)|}\{\sum_{i=1}^{t}\mu(\gamma_i)\}$.

Figure 3 represents the DF-execution order of the nodes of the multithreaded computation presented in Figure 2. Nodes are executed in the order determined by the number associated to them. Thus, the number of a node represents its DF-priority.

3.3 Scheduling Algorithms

A *scheduling algorithm* for a multithreaded computation decides which processors execute which instructions at every timestep. Roughly speaking, a scheduling algorithm is comprised by a set of rules to determine how processors execute threads. Thus, a scheduling algorithm takes as input the dynamic graph G and the number P of processors on which the computation should be executed. Since the graph is dynamic, only when a node is scheduled its outgoing edges in the computation dag are revealed, while a node is scheduled only if all its incoming edges in the dag have been revealed. An *online* algorithm bases its scheduling decision at each

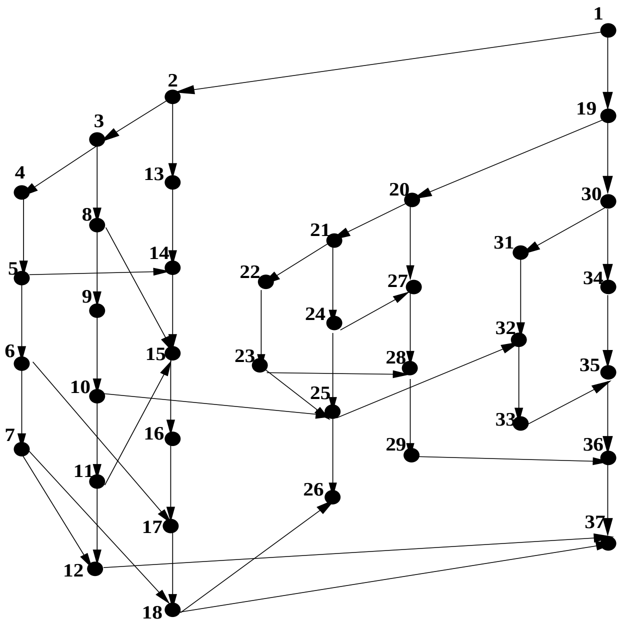

Figure 3: The DF-execution order of the multithreaded computation presented in the previous figure. The number associated to each node represents its DF-priority.

timestep on only the revealed nodes and edges of the dag. Thus, due to the dynamic nature of a multithreaded computation, the problem of scheduling efficiently such computations is intrinsically online.

A *work-stealing* algorithm works as follows. When a processor is idle, it becomes a *thief* and steals work from the local pool of a *victim* processor or from a centralized pool of work. The decision of which is going to be the victim processor (or which thread in the centralized pool will be stolen) may be taken either deterministically or in a randomized way, yielding to either deterministic or randomized algorithms, respectively.

If the algorithm works in a randomized way, the algorithm produces a probability distribution over all execution schedules $\mathcal{X}(G,P)$. For each such algorithm Alg, we denote by $T_{\mathsf{Alg}}(G,P)$ and by $S_{\mathsf{Alg}}(G,P)$ the random variables denoting the execution time and the space required, that is the *space complexity*, during the P-processor execution of a multithreaded computation G by Alg.

A *depth-first* scheduling algorithm assigns priorities to nodes/threads according to their serial execution order and concentrates on having the P processors always work on high priority nodes. All depth-first algorithms employ a centralized data structure of sets of ready nodes which are maintained in decreasing order of priorities. This data structure is called the *working data structure*. The set of nodes may contain from one single node each, to some subset of the nodes of a thread or even a set of nodes from individual threads. Each processor may work in at most one of these sets of nodes at each timestep (that is, each processor *owns* at most one of these sets). These sets are prioritized according to the serial execution order by choosing a *leading node* belonging to the set, which lends its priority to the entire set. A randomized work-stealing, depth-first scheduling algorithm works as follows. When a processor is idle it chooses *uniformly at random* a set of nodes among the highest priority sets in the working data structure and steals some node from this set. In this paper, we assume that an idle processor tries to steal a node from one of the $(1-\alpha)P$, where $0 < \alpha < 1$, highest priority sets in the working data structure (that is, it steals a node from the $(1-\alpha)P$ leftmost set of nodes in the working data strucutre).

3.4 Analytical Tools

Let $\mathcal{X}_{\mathsf{Alg}}(G,P)$ be the execution schedule produced by a work-stealing scheduling algorithm Alg for the multithreaded computation G on a P-processor parallel computer. A *batch* of nodes is the set of all nodes executed by some processor between consecutive steal attempts of the processor. Notice that all nodes of G can be partitioned into batches. However, since obtaining this partition depends on the way processors execute nodes, it might be different for different execution schedules. We call the first node executed in a batch a *heavy* node, while all other nodes of a batch are called *light* nodes. For the purpose of our analysis, we assume that the heavy node allocates the whole amount of memory required by a batch, while light nodes do not allocate any memory.

Consider a P-schedule $\mathcal{X}(G,P)$ and let $t \in [\,|\mathcal{X}(G,P)|\,]$ be any timestep. We denote by $\mathcal{X}(G,P) \uparrow t$ the prefix of $\mathcal{X}(G,P)$ consisting only of the first t timesteps of $\mathcal{X}(G,P)$; that is, $\mathcal{X}(G,P) \uparrow t = \mathcal{X}_1(G,P), \ldots, \mathcal{X}_t(G,P)$. We call $\mathcal{X}(G,P) \uparrow t$ the *parallel prefix of $\mathcal{X}(G,P)$ at timestep t*. We denote by $(\mathcal{X}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$ the longest sequence of nodes in $\mathcal{X}_{DF}(G)$ belonging in some $\mathcal{X}_i(G,P)$, where $i \in [t]$. We call $(\mathcal{X}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$ the *longest serial prefix of $\mathcal{X}(G,P)$ at timestep t*. The *set of non-premature nodes at timestep t* in an execution schedule $\mathcal{X}(G,P)$ are all nodes in $(\mathcal{X}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$. The *set of premature nodes at timestep t* in $\mathcal{X}(G,P)$ consists of all nodes γ such that $\gamma \notin (\mathcal{X}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$, but there exists some integer $i \in [t]$ such that $\gamma \in \mathcal{X}_i(G,P)$. Roughly speaking, premature nodes are all nodes that have been executed by t in the P-processor execution schedule $\mathcal{X}(G,P)$, but do not belong to the longest serial prefix of $\mathcal{X}(G,P)$ at timestep t.

For each node $\gamma \in G$ we define its *last parent* (in $\mathcal{X}(G,P)$) to be the last executed parent of γ in $\mathcal{X}(G,P)$. If there are more than one such nodes then the last parent of γ is found as follows. Let t be the timestep that these nodes are executed. If γ is placed in the set of nodes owned by the processor executing one of these parents or if the thread 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. Otherwise, choose the last parent of γ to be any arbitrary node among its last executed parents.

Consider any P-processor parallel execution schedule $\mathcal{X}(G,P)$ and let γ be any node of G. We define the *critical path of* γ, denoted $\mathcal{P}_\gamma$, as follows. $\mathcal{P}_\gamma$ is initialized to γ; for each node $\gamma' \in \mathcal{P}_\gamma$ the last parent of γ' is added to $\mathcal{P}_\gamma$. Notice that for each node γ, the root node must belong to $\mathcal{P}_\gamma$ and thus $\mathcal{P}_\gamma$ 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. Then, if $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of node γ, for each $i \in [\delta(\gamma)]$, node γ_i is the node of depth i in path $\mathcal{P}_\gamma$. Moreover, γ_1 is the root node, while $\gamma_{\delta(\gamma)} = \gamma$.

Assume that t is the timestep that some node $\gamma \in G$ is executed and let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ. Denote by t_i the timestep that node γ_i is executed, where $i \in [\delta(\gamma)]$. Since $\mathcal{P}_\gamma$ 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 $i \in [\delta(\gamma)-1]$, $t_i < t_{i+1}$. With each node $\gamma \in G$ we associate a set of intervals, which we call the *set of critical intervals of* γ, and we denote

$\mathcal{I}(\mathcal{P}_\gamma) = \{I_1, \ldots, I_{\delta(\gamma)}\}$. For each $i \in [\delta(\gamma)]$, we define I_i as follows. Let $I_1 = \{1\}$; for each $i = 2, \ldots, \delta(\gamma)$, $I_i = \{t_{i-1}+1, \ldots, t_i\}$. Since $\mathcal{P}_\gamma$ is a path from the root to node γ, the critical intervals of γ cover the duration of execution $1, \ldots, t_{\delta(\gamma)}$. Notice that for each $i \in [\delta(\gamma)]$, γ_i is a ready node for all timesteps in the critical interval I_i.

Let Alg be any randomized, work-stealing, *depth-first* algorithm and consider any execution schedule $\mathcal{X}_{\mathsf{Alg}}(G, P)$ that is produces. For any timestep $t \in [|\mathcal{X}_{\mathsf{Alg}}(G, P)|]$ and any ready node $\gamma \in G$ at t, we say that t is a *candidate timestep for* γ, if the set of nodes in which γ belongs is one of the target sets for steal attempts at t. Let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ and $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ be the set of critical intervals of γ. Let $t_\gamma \in [|\mathcal{X}_{\mathsf{Alg}}(G, P)|]$ be the timestep that γ is executed. Recall that the set of intervals $\mathcal{I}_\gamma$ covers all timesteps $\{1, \ldots, t_\gamma\}$. For each $i \in [\delta(\gamma)]$, we denote by C_i the *set of candidate timesteps for* $\gamma_i \in \mathcal{P}_\gamma$. Clearly, $C_i \subseteq I_i$. Let $\mathcal{C}_\gamma = \{C_1, \ldots, C_{\delta(\gamma)}\}$. We can partition all timesteps in each of the C_i, where $i \in [\delta(\gamma)]$ 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 one of the following events occurs:

(1) node $\gamma' \in \mathcal{P}_\gamma$ for which timestep t is critical is executed till timestep t';

(2) one of the $\Theta(P)$ steal attempts initiated in R targets the ready list containing γ'.

It can be easily proved that a round is successful with probability at least $1/2$, as stated by the following lemma[3].

Lemma 3.1 (Narlikar) *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{Alg}}(G, P)$ is any execution schedule produced by any randomized, work-stealing, depth-first scheduling algorithm* Alg. *Consider any node $\gamma \in G$, let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ, $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ be the set of critical intervals of γ and $\mathcal{C}_\gamma$ be the sets of candidate timesteps during $\mathcal{I}_\gamma$. Then, the probability that any round of steal attempts in $\mathcal{C}_\gamma$ is successful is at least $1/2$, independently of other rounds.*

For depth-first scheduling algorithms it can also be proved that if for two nodes it holds that the one is a non-premature node at some timestep and the other is a premature node at the timestep, then any previous candidate timestep of the premature node, in which the non-premature node is ready in the working data structure, is also a candidate timestep for the non-premature node.

Proposition 3.2 (Narlikar) *Consider any multithreaded computation G and let $P > 0$ be any integer. Consider any execution schedule $\mathcal{X}_{\mathsf{Alg}}(G, P)$ and any integer $t \in [|\mathcal{X}_{\mathsf{Alg}}(G, P)|]$. Let γ be any non-premature node at t and let γ' be some premature node at t. Then, for any timestep $t' \in [t]$, if t' is a candidate timestep for γ' and γ is ready in the working data structure at t', then t' is a candidate timestep for γ.*

4 The Algorithm PSDF

In this section we present a detailed description of the depth-first, work-stealing, scheduling algorithm PSDF.

[3] Because targets for steals are chosen uniformly at random from the leftmost P sets of nodes, and because each phase has at least P steals, the probability that a round is successful is $\geq 1-(1-1/P)^P \geq 1-1/e > 1/2$.

Since PSDF is a depth-first scheduling algorithm, the nodes are prioritized according to their serial execution order. All active threads are organized in a set of doubly-ended lists. Each list contains multiple threads with contiguous priorities. Threads in bottommost positions of a list have larger priorities than threads in topmost positions.

Algorithm PSDF uses a 2-3 (priority) tree $\mathcal{B}$ of lists of threads, which is called the *working tree*. The lists reside at the leaves of the tree and they are prioritized according to the priority of the topmost thread they contain. Each list contains threads of larger priority than any other list on its right. A list might be in *ready* or in *stalled* state. All lists, independently of their states, are stored according to their priorities in the working tree. Instead of keeping keys, each internal node of the 2-3 tree contains counters, which represent the number of *ready* lists in the subtree rooted at the node. The working tree supports the following operations, which are called the *priority tree operations*: 1) given a set of leaf elements: 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 element belonging to the set, and 2) find the P leftmost ready lists in the working tree. The *size* of the working tree is the number of leaves that it contains.

The algorithm 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 a constant fraction αP of the processors, where $0 < \alpha < 1$, while the remaining $(1-\alpha)P$ processors are the working processors. All processors run in parallel; the working processors run asynchronously with each other and with the scheduling processors, while the scheduling processors run synchronously.

Algorithm PSDF maintains a set of arrays, which is called the *set of working arrays*. The first array is a one dimensional array A of length $(1-\alpha)P$, which is called the *active array*. The active array is periodically updated to maintain pointers to the $(1-\alpha)P$ leftmost ready lists in $\mathcal{B}$. The *deletion array*, denoted by Δ, is a two-dimensional array with 3 rows and $(1-\alpha)P$ columns. For each column $i \in [(1-\alpha)P]$, there exists a toggle bit δ_i. The toggle bit can be either 0 or 1 and plays an important role in accessing Δ. Roughly speaking, the deletion array contains pointers to lists that they are to be deleted from the working tree. The *stall array*, denoted by Π, is a two dimensional array with 3 rows and $(1-\alpha)P$ columns. As with the deletion array, for each column $i \in [(1-\alpha)P]$, there exists a toggle bit π_i. Roughly speaking, the stall array contains pointers to ready lists that contain stalled (suspended) threads and thus special actions should be taken for these lists in the working tree. The *steal array*, denoted by Σ, is a two-dimensional array with 2 rows and $(1-\alpha)P$ columns, holding pointers to newly created ready lists that should be inserted in the working tree. For each column $i \in [(1-\alpha)P]$ of the steal array, there exists a toggle bit s_i. The last array that the algorithm uses is the *enabling array*, denoted by E. The enabling array is a two dimensional array of 3 rows and $(1-\alpha)P$ columns, containing pointers to lists of synchronization variables. Roughly speaking, whenever a working processor $p \in [(1-\alpha)P]$ writes a variable with waiting threads, the processor adds a pointer to the variable in a list pointed to by some element of E. There exists an amount g, which is called the *enabling threshold*, such that P^g is an upper bound on the number of synchronization variables with suspended threads that a processor is allowed to write per scheduling iteration while working on the same ready list[4]. The enabling threshold can be a user-determined parameter, but we consider it a constant greater or equal to one for the purpose of our algorithm's analysis. For

[4]Notice that a processor may work on more than one ready list per scheduling iteration.

each column $i \in [(1-\alpha)P]$ of E, there exists a toggle bit ζ_i, as well as a flag θ_i. Moreover, for all integers i, j, $1 \leq i \leq 3$, $1 \leq j \leq (1-\alpha)P$, there exists a counter, denoted $sync\_count[i][j]$, associated to $E[i][j]$, which counts the number of synchronization variables contained in the list pointed to by $E[i][j]$. All elements in each of these lists of array E are allocated by a contiguous chunk of memory. Initially, all arrays are empty and all toggle bits, flags and counters are 0. The toggle bits change their states from 0 to 1 and vice versa every time they are accessed.

The algorithm uses also a set of *synchronization queues* (see Section 2.4), one per each synchronization variable, in order to keep all suspended threads on each such variable. The synchronization queues are accessed only by the scheduling processors.

The working processors access only the working arrays described above, and have no access to the working tree or the synchronization queues. The scheduling processors have access to all data structures. The working processors synchronize and communicate with the scheduling processors only through the set of common data structures (that is, the arrays) they share.

Figure 4 presents all data structures used by the algorithm, while Figure 5 presents the lists of threads and their decreasing order of priorities as maintained by algorithm PSDF.

Each working processor $p \in [(1-\alpha)P]$ maintains two integers h_p and λ_p, called the *heating index* and the *enabling index*, respectively. Initially, all the working processors are idle and all the heating and enabling indices are NULL. An idle processor performs stealing in order to attain a ready list. It selects an integer $m \in [(1-\alpha)P]$ uniformly at random and tries to steal the topmost thread from the ready list pointed to by the mth leftmost element of the array A. If a steal attempt on the list pointed to by the mth element of A is successful, the processor does the following. It first stores m in its heating index variable; that is, the heating index stores the value of the index in array A from which the processor achieved to attain work for the last time. Then, the processor stores ζ_m to its enabling index variable and negates ζ_m; that is, λ_p contains the value of the toggle bit of column h_p of array E at the timestep of the stealing. If the stolen thread is the last thread in the list and the list is not owned by any processor the processor becomes the owner of the list and starts executing the stolen thread. If the stolen thread is not the last thread in its list, or if the list is owned by some other processor, the stealing processor creates a new ready list for itself. The processor then adds a pointer to this ready list in $\Sigma[s_{h_p}+1][h_p]$ entry of the steal array Σ, negates s_{h_p}, and starts executing the stolen thread. When a processor achieves to attain work, we say that it *owns* the ready list on which it works. At any time a processor owns at most one ready list. The heating and the enabling indices of a processor are updated every time a steal attempt initiated by the processor succeeds.

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 $m \in [(1-\alpha)P]$ independently and uniformly at random, etc.

Each working processor assigns a memory quota of K bytes from which each thread in the ready list owned by the processor can allocate heap and stack data. Each time a processor steals, its memory quota is re-assigned to K.

When a working processor achieves to attain work (that is, when its steal attempt succeeds), it starts executing the stolen thread without preemption until one of the following events happens: (1) it forks a child thread, (2) it suspends waiting on some synchronization

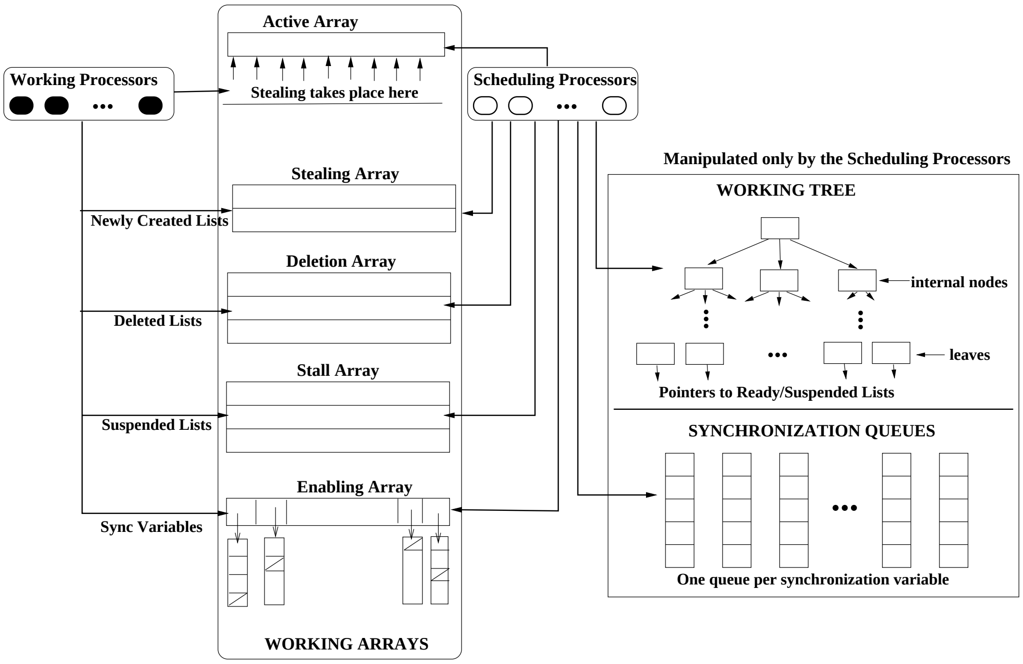

Figure 4: Data structures used by the Algorithm PSDF. The working processors access only the working arrays. The working tree and the synchronization queues are manipulated only by the scheduling processors. Stealing is performed on the active array, which contain pointers to the $(1-\alpha)P$ leftmost ready lists in the working tree. Information about newly created lists due to stealing is kept in the steal array, while pointers to the lists that are to be deleted are maintained in the deletion array. The stall array stores information about suspended threads, while in the enabling array pointers to lists of written synchronization variables with waiting threads are stored. The elements of all these arrays are accessed periodically by the scheduling processors, so that appropriate modifications to be done to the working tree, as well as to the synchronization queues.

edge, (3) it terminates, or (4) it runs out of its memory quota.

When a thread forks a child thread, the processor pushes back the thread that performed the spawning on the bottommost position of its ready list and continues work on the child thread. When a thread suspends, the processor (let it be p) takes the following actions. It first checks if $\Pi[3][p]$ is empty and if yes it inserts a pointer in $\Pi[3][p]$ pointed to the corresponding leaf of the working tree. If $\Pi[3][p]$ is not empty, the processor modifies the $\Pi[\pi_{h_p}+1][h_p]$ element of the array Π by adding a pointer to the leaf of the working tree pointing to the suspended list[5], and negates the toggle bit π_{h_p}. If the flag θ_p equals 0, the processor sets θ_p to 1. When a thread terminates, the processor checks the ready list that it owns and if it is non-empty, it removes the bottommost thread of it and starts work on it. If the ready list is empty, the processor marks the ready list as deleted by performing the following actions. It first checks if $\Delta[3][p]$ is empty and if yes it inserts a pointer in $\Delta[3][p]$ pointed to the corresponding leaf of the working tree. If $\Delta[3][p]$ is not empty, the processor inserts a pointer

[5]Notice that h_p is the heating index of processor p and π_{h_p} is the toggle bit of the h_p row of array Π.

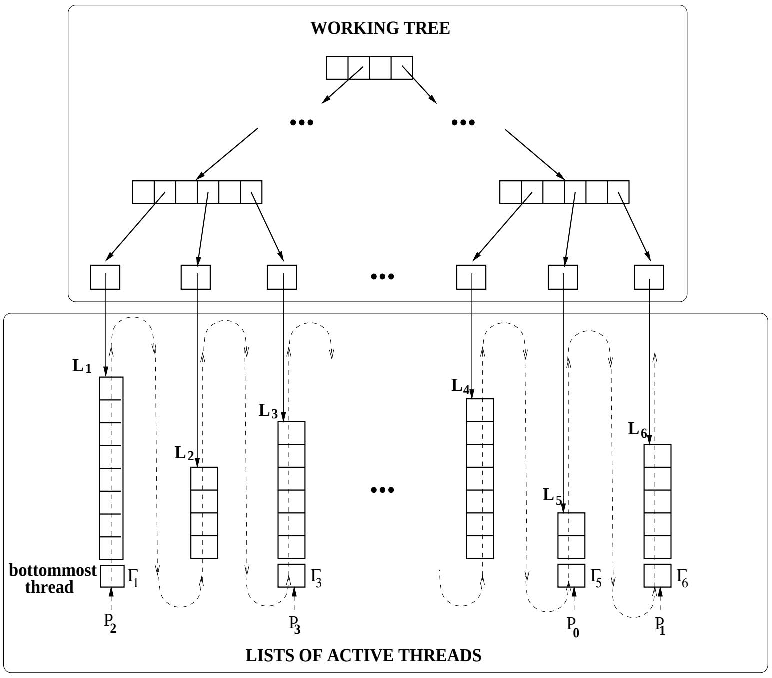

Figure 5: Lists of active threads maintained by PSDF. The dotted curved line presents the decreasing order of priorities of threads in the lists. We can see 6 lists in the system, namely L_1 to L_6. Only the lists L_1, L_3, L_5 and L_6 are owned by some processor (in our figure, these processors are p_2, p_3, p_0 and p_1, respectively), while all other lists are not owned by any processor. 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 should be 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. Notice that the leaves of the working tree contain pointers to the lists of active threads in the system.

to the leaf of the working tree pointing to the suspended list in $\Delta[\delta_{h_p}+1][h_p]$ element of the array Δ, and negates the toggle bit δ_{h_p}. If the flag θ_p equals 0, the processor sets θ_p to 1. Roughly speaking, the working processors manipulates the arrays Π and Δ as follows. Each processor p has unique access to $\Pi[3][p]$ (or $\Sigma[3][p]$ respectively); that is, each processor is allowed to access only one element of the third row of the array Π (Σ). Thus, each processor tries to store its data about suspended (deleted) lists in its associated element of Π (Σ). 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 heating index and the row determined by the toggle bit of this column. The processor finally sets θ_p to declare the fact that it has already worked on some ready list in this "scheduling iteration".

Special actions should be taken when a processor p executes an instruction that writes on a synchronization variable with waiting threads. If θ_p equals 0, the processor modifies the list $E[3][p]$ by adding a pointer to the synchronization queue of the written synchronization variable and increases by one the counter $sync\_count[3][p]$. If $sync\_count[3][p]$ becomes P^g

(where g is the enabling threshold), the processor sets θ_p to 1. Moreover, the processor gives up its ready list and starts work-stealing in order to attain a new ready list. If θ_p equals 1, the processor inserts a pointer to the synchronization queue of the written variable in list $E[\lambda_p + 1][h_p]$. Moreover, it increases $sync\_count[\lambda_p + 1][h_p]$ by 1. If $sync\_count[\lambda_p + 1][h_p]$ becomes P^g, the processor gives up its ready list and performs stealing in order to attain a new ready list.

Algorithm PSDF employs the same strategy as any other depth-first scheduling algorithm in order to cope with large allocations. Thus, if a thread contains an action that needs more memory than the assigned memory quota to the ready list, e.g. the total memory required is $m > K$, then $\lfloor m/K \rfloor$ dummy threads are inserted before the allocation. The dummy threads perform a no-op operation each, but 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. Clearly, each one of the $\lfloor m/K \rfloor$ dummy threads is allowed to allocate K space, which it does not actually allocate. Thus, by executing the instruction that performs the large allocation after all these dummy threads have been executed, is compatible to our space quota concept.

We continue to describe the actions taken by the scheduling processors. Recall that all scheduling processors work in parallel in a synchronous way. As long as there exist threads in the system, the scheduling processors execute a loop containing the five tasks described below:

(1) *Splitting Task.* For each element in the steal array Σ, they 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 [6].

(2) *List Suspension Task.* For each element in the stalling array Π, the scheduling processors take the following actions. If the list pointed to by the element contains only the suspended thread, the list is marked as suspended and the counters up to the root of the working tree are updated. If the list contains more threads, then the corresponding element of the working tree 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 list pointed to by the initial element before splitting. The counters up the working tree are updated. An element pointing to the suspended list is added in the synchronization queue of the corresponding synchronization variable.

(3) *Deletion Task.* For each element in the deletion array Δ, delete the corresponding element of the working tree.

(4) *Enabling Task.* All synchronizations queues pointed to by elements of the lists of the enabling array contain enabled threads. There are $3(1-\alpha)P$ lists of synchronization variables and for each such variable there exists a synchronization queue of enabled threads. The scheduling processors execute the following algorithm. They first perform a prefix sums computation on the "array" of the `sync_count` counters and they create an array of partial sums. Then, they zero up all counters `sync_count` and they update

[6] Pointers to these lists are stored in array Σ.

all elements of E to NULL, and all flags θ to 0. After the scheduling processors has determined the total number of synchronization variables in the lists of E, they can divide the elements into sets of the same size on which each one of them should work (access and process the elements that the set contains). Each processor can find in which lists these elements reside by executing a BinarySearch procedure (see e.g., in [12]) on the array containing the prefix sums twice (one for its starting processing element and one for its ending processing element). The processors then create a new array with as many elements as synchronization variables in the lists of E containing the number of the enabled threads that the synchronization queue for each such variable contains. The scheduling processors apply again the same procedure in order to find the sequence of elements in these synchronizations queues that each one of them should work on (they thus perform a prefix sums computation and then each one of them executes a BinarySearch procedure twice on the array containing the new prefix sums. Finally, each one of the processors accesses one by one the enabled threads that it has to process and changes the state of the corresponding list in the working tree from suspended to ready, updating the counters up to the root of the tree.

(5) *Activation Task.* Find the P leftmost ready lists in the working tree and update the elements of the active array A to point to these lists.

As long as there exist threads in the system, the scheduling processor execute a loop containing the five tasks described above. We call each iteration of this loop a *scheduling iteration.* Consider any element e of some of the arrays A, Σ, Π, Δ, or E. We say that a scheduling iteration *updates* e if the scheduling processors process the information stored in e (if any), 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'; that is, t' is the first timestep after t that e is updated again. Then, the time interval (t, t') is called *scheduling interval of* e. Notice that e may be written only by working processors during its scheduling interval.

We prove later in our analysis that each scheduling iteration i needs $O((E_i/P+1)\log(|\mathcal{B}_i|+P))$ timesteps to be executed, where E_i are the threads which are enabled during the scheduling iteration and $|\mathcal{B}_i|$ is the size of the working tree at the beginning of the scheduling iteration. The scheduling processors count the number of timesteps that have elapsed during the execution of each scheduling iteration and if it is less than $3\log(|\mathcal{B}_i|+P))$ they wait till $3\log(|\mathcal{B}_i|+P))$ timesteps have elapsed before proceeding to the next scheduling iteration[7].

We allow only one steal attempt to take place at each ready list during a scheduling iteration; that is, only one steal attempt can be performed on any list between consecutive scheduling iterations. Thus, we associate with each list 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 scheduling processors re-initiate to 0 the stealing bits of lists that they index during the activation task. All steal attempts to ready lists with their stealing bits to 1 fail. A steal attempt may also fail if more than one steal attempts target the same ready list. In this case we assume that only one of these steal attempts is successful, while all other steal attempts fail. Additionally, if the number of ready

[7]This delay allows the working processor to perform a satisfactory number of steal attempts on the elements of the active array per scheduling iteration. Although introducing this delay is not necessary for proving our time bound in Section 5.5, it appears to simplify our analysis.

lists is less than the size of the active array, all steal attempts directed to empty slots of the active array fail.

5 Analysis

In order to simplify our analysis, we assume that the heavy node of a batch allocates the whole amount of memory required by all nodes in it, while light nodes do not allocate any memory. Moreover, we assume that all nodes allocate at most K memory. The case of larger allocations and how they are treated by our algorithm is discussed in Section 6.

5.1 Cost Model

Our cost model closely follows the one presented in [26, Section 6.3.1].

As with all previous work in this research area [1, 5, 6, 7, 9, 25, 26] we assume that the timesteps are synchronized across all the processors.

If multiple processors select the same ready list for stealing then contention may arise. We assume a non-blocking implementation of the ready lists, like the one presented by Arora *et al.* in [1]. Thus, if a number of steal attempts target the same list at a timestep, 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 fail the processor tries again in the next timestep.

As in [25], we assume that an allocation of M bytes of memory has a depth of $O(\log M)$. This assumption is reasonable in systems with binary forks that zero out the memory as soon as it is allocated; this zeroing is usually performed in parallel by forking a tree of height $\Theta(\log M)$.

5.2 Properties of the Working Arrays

Our first lemma is immediate due to the way the scheduling processors work. The scheduling processors execute a loop. In each iteration of the loop the five scheduling tasks are executed. This and the definition of the scheduling interval of any position e of the working arrays immediately imply:

Lemma 5.1 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF *and let e be some arbitrary position in any of the arrays Σ, Π, Δ and E. Then, exactly one activation task is executed during any scheduling interval of e.*

We continue to prove that at most two steal attempts can occur in the scheduling interval of e.

Proposition 5.2 *Consider any multithreaded computation G, let $P > 0$ be any integer and let α, $0 < \alpha < 1$, be any real. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF *and let e be some arbitrary position in any of the arrays Σ, Π, Δ and E. Then, if $m \in [(1-\alpha)P]$ is any integer, it holds that at most two* successful *steal attempts target $A[m]$ during any scheduling interval of e.*

Proof: Let e be any position in some of the arrays Σ, Π, Δ and E. Assume that (t, t') is some scheduling interval of e. Algorithm PSDF allows only one successful steal attempt to target $A[m]$ at each scheduling iteration. By Lemma 5.1 only one activation task is executed during the scheduling interval. Thus, there exist exactly one timestep t_a such that $t < t_a < t'$ and $A[m]$ is updated at t_a, so that in the worst case, at most two steal attempts may succeed on $A[m]$, one in the time interval (t, t_a) and one in the time interval (t_a, t'), as needed. ∎

We next prove that the size of the working arrays is large enough to hold all required information produced by the working processors at each scheduling iteration, so that information for updates on the working tree is never lost (e.g., because the working processors may overwrite some of the elements of these arrays) in the same scheduling iteration.

Proposition 5.3 *Consider any multithreaded computation G, let $P > 0$ be any integer and α, $0 < \alpha < 1$, be any real. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF. *Then, for any position e in any of the arrays Σ, Π, Δ, and E, it holds that e is written only once by the working processors during any of its scheduling intervals.*

Proof: We start by proving that no position of the steal array Σ is written twice during any of its scheduling intervals. Let $\Sigma[i][j]$ be any position of Σ, where $1 \leq i \leq 2$ and $1 \leq j \leq (1-\alpha)P$. Assume that (t, t') is some scheduling interval of $\Sigma[i][j]$. We will prove that $\Sigma[i][j]$ can be written only once in (t, t'). Clearly, whenever column j is indexed the toggle bit s_j changes state from 0 to 1 and vice versa. Thus, it suffices to prove that column j is indexed at most twice in the time interval (t, t'). The only case in which column j is accessed by some processor p is if the processor achieves to steal some thread from the list pointed by $A[j]$. However, Proposition 5.2 implies that at most two successful steal attempts may target $A[j]$ during the scheduling interval (t, t'). In the worst case, each of these two steal attempts need to update one entry of column j of array Σ. However, due to the toggle bit s_j these two updates are performed in different elements of column j.

We continue to prove that no position of the stall array Π is written twice during any of its scheduling intervals. Proving that no position of the deletion array Δ is written twice during any of its scheduling intervals can be done in an analogous way.

Let $\Pi[i][j]$ be any position of Π, where $1 \leq i \leq 3$ and $1 \leq j \leq (1-\alpha)P$. Assume that (t, t') is some scheduling interval of $\Pi[i][j]$. We proceed by case analysis. Assume first that $i = 3$. Recall that $\Pi[3][j]$ can be written only by the working processor j and only once per each scheduling interval of $\Pi[3][j]$. Thus, the stated claim holds. Assume now that $i < 3$. Proposition 5.2 implies that at most two steal attempts may succeed on $A[j]$. Assume that processors p_1 and p_2 perform each one of these two steal attempts, respectively. In the worst case, both stolen lists suspend and both $\Pi[3][p_1]$, $\Pi[3][p_2]$ are occupied, so that both processors will access the jth column of Π and add a pointer pointing to the suspended lists there. However, due to the toggle bit π_j these two updates will be performed in different positions of column j.

It is worth pointing out how it can occur that all three elements of a column are written. This is the case when some processor p works on some ready list which it attained during some scheduling iteration in the past. If this list suspends during the scheduling interval (t, t'), information about this suspended list must be written in array Π. This event can only happen before processor p starts stealing. Since the processor started working on this ready list at some scheduling iteration in the past, $\Pi[3][j]$ is empty and the processor can always use $\Pi[3][j]$ to write this sort of information. Then, it may happen that two successful steal attempts take place on $A[p]$ by processors that they have already written their own positions in $\Pi[3]$, both of them leading to lists which later suspend. In this case, all elements of column p of Π will be written once during the scheduling iteration.

We next prove that no element of the enabling array will be updated twice during any of its scheduling intervals; that is, each element of E always points to the same list during any of its scheduling iterations (although new elements may be added in this list). Let $E[i][j]$ be any position of E, where $1 \leq i \leq 3$ and $1 \leq j \leq (1-\alpha)P$. Assume that (t, t') is some scheduling interval of $E[i][j]$. We proceed by case analysis. Assume first that $i = 3$. Recall that $E[3][j]$ can be written only by the working processor j. Due to the way flag θ_j is set to 1, it cannot happen that processor j updates $E[3][j]$ twice in the same scheduling interval of $E[3][j]$, as needed. Assume now that $i < 3$. Proposition 5.2 implies that at most two steal attempts may succeed on $A[j]$. Assume that processors p_1 and p_2 perform each one of these two steal attempts, respectively. In the worst case, both stolen lists contain threads that write synchronization variables with waiting threads and both θ_{p_1}, θ_{p_2} are set to 1, so that both processors will access the jth column of E. However, due to the way the enabling indices λ_{p_1} and λ_{p_2} are updated, it holds that $\lambda_{p_1} \neq \lambda_{p_2}$, so that these two updates will be performed in different positions of column j, as needed.

At this point the proof of Proposition 5.3 is complete. ∎

We finally prove that each list pointed to by any element of the enabling array cannot contain more than P^g elements, where g is the enabling threshold of PSDF, so that the total number of elements in all these lists is $O(P^{g+1})$.

Proposition 5.4 *Consider any multithreaded computation G, let $P > 0$ be any integer and α, $0 < \alpha < 1$, be any real. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF. *If g is the enabling threshold of* PSDF, *then the total number of elements contained in the lists of synchronization variables pointed to by elements of E is $O(P^{g+1})$.*

Proof: Proposition 5.3 implies that at most $3(1-\alpha)P$ lists of synchronization variables (one per each element of E) are created at each scheduling iteration. Clearly, each one of these lists contains pointers to synchronization variables with waiting threads which have been written by threads belonging to the same ready list. Due to the enabling threshold, each such list contains at most P^g elements, so that the total number of elements in all these lists is at most $3(1-\alpha)P\,P^g \in O(P^{g+1})$, as needed. ∎

5.3 Properties of the Working Set

Our first proposition states useful properties on the ordering of threads residing in the working tree.

Proposition 5.5 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF. *Then, the following properties on the ordering of threads in lists are maintained during the execution of $\mathcal{X}_{\mathsf{PSDF}}(G, P)$:*

(1) *threads in each list are in decreasing order of priorities from bottom to top;*

(2) *a thread currently executing on a processor has higher priority than all other threads on the processor's ready list;*

(3) *the threads in any given list in $\mathcal{B}$ have higher priorities than all other threads in all lists to its right in $\mathcal{B}$.*

Proof: By induction on the execution time. For the basis case where $t = 0$ the claim holds vacuously, since the only ready thread is the root thread. Assume that the three properties hold at the start of some timestep t, such that $t \geq 0$. We will prove that the properties hold at the end of t.

The following events may lead to modifications on some list: (1) a thread forks a child thread, (2) a thread terminates, (3) a thread suspends, (4) a thread is enabled, (5) a thread is preempted due to exhaustion of the processor's memory quota, (6) a processor steals the topmost thread of some list, (7) a list is deleted from $\mathcal{B}$, and (8) a list is inserted in $\mathcal{B}$. We consider each of the above cases separately.

We start by considering the case that a thread forks a child thread. By induction hypothesis the parent thread has the higher priority than any other thread in its list. However, the child thread, which is placed in the bottommost position of the list, has higher priority than its parent, while the processor continues execution on the child thread. Thus, properties (1) and (2) hold. Moreover, by induction hypothesis all threads in the list have higher priorities than any thread in lists to its right. Since the child thread has the highest priority among all threads in its list, property (3) also holds.

We next consider the case that a thread terminates. If the ready list of the processor is non-empty the processor starts work on the bottommost thread of its ready list. By induction hypothesis, threads in the ready list are in decreasing order of priorities from bottom to top. Thus, the bottommost thread becomes now the thread with the highest priority among all threads in its list, so that property (2) holds. All other properties hold by induction hypothesis. If the ready list is empty, the ready list is deleted. The case of a list deletion is considered later in our proof.

We continue to consider the case that a thread suspends. The suspended thread is extracted from the ready list. Clearly, the suspended thread is the bottommost thread in this list, so that by induction hypothesis all properties hold for this list. The suspended thread is the only thread in a new list which is later inserted in the working tree. Clearly, properties (1) and (2) for the newly created list hold vacuoucly. The case of a list insertion in the working tree is considered later in our proof.

We next consider the case that a thread is enabled. The only change in the working tree is that the state of some lists may change from stalled to ready. Thus, by induction hypothesis, all properties hold.

When a thread is preempted due to exhaustion of its memory quota, it is put back on the bottom of its list and the processor starts work on a dummy child thread. Clearly, the preempted thread is the highest priority thread among those in its processor's ready list and all its child threads (independently of whether they are dummy or not) have higher priority. This and induction hypothesis imply that all properties hold.

We next consider the case that a processor steals the topmost thread of some list (which we call here target list). The removal of the topmost thread from the target list does not affect the validity of the three properties of the list. The thread may be stolen from a processor's list while one of the above events occurs on the processor itself. This does not affect the validity of the properties. The stolen thread is placed in a new ready list which is later inserted in the working tree. The case of a list insertion in the working tree is considered later in our proof.

We next consider the case that a list is deleted from $\mathcal{B}$. Clearly, the list is empty. By induction hypothesis all properties hold.

We finally consider the case that a new list is inserted in the working tree. The new list is created in one of the following ways: (1) by stealing, or (2) by suspension of some thread. We proceed by case analysis. Assume first that the new list was created by stealing. The stolen thread is the topmost thread of some list (target list) and the newly created list is placed on the immediate right of the target list in the working tree. By induction hypothesis, the stolen thread has lower priority than all other threads in the target list but higher priority than all threads in lists on the right of the target list. All other threads residing in the list are descendants of the stolen thread and thus they reside in positions closer to the bottommost position of the list. By induction hypothesis, all threads in the new list are in decreasing order of priorities from bottom to top. Thus, all these threads have higher priorities than the stolen thread, so that all threads in the new list have higher priorities than all other threads in lists of B in its right. Therefore, by placing the newly created list on the immediate right of the target list, property (3) holds. All other properties hold for the newly created list either vacuously or by induction hypothesis. Assume now that the new list contains a suspended thread. The suspended thread is the bottommost thread of some list (which we call here the source list). Thus, by induction hypothesis it has the higher priority than all other threads contained in the list. Moreover, all lists on the left of the source list contain threads with higher priorities than this thread. Recall that the newly created list is placed on the immediate left of the source list. Thus, all properties hold.

This completes the proof of the proposition. ∎

We now explore an interesting relation among threads of the same list. Proposition 5.6 generalizes a corresponding claim, proved in [26, Lemma 6.2,Chapter 6] for algorithm DFDeques. This claim states that similar properties to those presented below are maintained by DFDeques, when node γ is the last non-premature node to be executed in the first t timesteps of DFDeques execution. Indeed, similar arguments to those presented in [26] can be used to prove the properties in the more general form presented below. However, the fact that our algorithm treats more general computations than the algorithm presented in [26] complicates our proof.

Proposition 5.6 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by* PSDF. *Consider any node $\gamma \in G$ and let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ. Then, for any $i \in [\delta(\gamma)]$,*

(1) *if there exists some timestep t such that γ_i is the ready node of a thread in some ready list*[8]*, then every ready node above γ_i in its ready list is the right child of some node in $\mathcal{P}_\gamma$;*

(2) *when γ_i is executed on a processor, every ready node in the processor's ready list is the right child of some node in $\mathcal{P}_\gamma$.*

[8]Notice that the ready list that γ_i resides might have not yet been inserted in the working tree.

Proof: By induction on i.

Basis Case

For the basis case, where $i = 1$, recall that γ_1 is the root thread. Initially, it is the only thread in its list and gets executed before any new nodes are created. Thus, the claim holds trivially.

Induction Hypothesis

Assume that for any i, $1 < i \leq \delta(\gamma)$, the following hold for all j such that $1 \leq j < i$:

(1) if there exists some timestep t such that γ_j is the ready node of a thread in some ready list, then every ready node above γ_j in its ready list is the right child of some node in $\mathcal{P}_\gamma$;

(2) when γ_j is executed on a processor, every ready node in the processor's ready list is the right child of some node in $\mathcal{P}_\gamma$.

Induction Step

We prove that the claim holds for γ_i; that is,

(1) if there exists some timestep t such that γ_i is the ready node of a thread in some ready list, then every ready node above γ_i in its ready list is the right child of some node in $\mathcal{P}_\gamma$;

(2) when γ_i is executed on a processor, every ready node in the processor's ready list is the right child of some node in $\mathcal{P}_\gamma$.

By definition of the critical path, node γ_{i-1} is the last parent of node γ_i. Thus, γ_i becomes ready immediately after γ_{i-1} is executed on some processor. There are the following three possibilities: (1) γ_i is executed immediately following γ_{i-1} on that processor, (2) the thread containing γ_i is placed in the processor's list after γ_{i-1} is executed, and (3) the thread containing γ_i is stalled and the execution of γ_{i-1} enables it. We consider each of these three possibilities separately.

Assume first that γ_i is executed immediately following γ_{i-1} on that processor. Since γ_i is never placed into a list property (1) holds vacuously. The only change that may occur to the ready nodes in the list before the execution of γ_i is that the right child of γ_{i-1} becomes ready. This can happen only if γ_{i-1} was a fork with γ_i as its left child. Since the new node is the right child of some node in $\mathcal{P}_\gamma$, in any case (using induction hypothesis), property (2) holds.

Assume now that the thread containing γ_i is placed in the processor's list after γ_{i-1} is executed. This event may happen in three cases: (1) node γ_{i-1} executes a spawn and γ_i is its right child, (2) the processor exhausted its memory quota, or (3) the processor exhausted its enabling quota. Assume first that γ_i is the right child of γ_{i-1}. Clearly, γ_i belongs to the same thread as γ_{i-1}, and no thread can be added above this thread in the ready list before the execution of γ_{i-1}. Thus, by induction hypothesis the properties hold. Assume next that the processor exhausted its memory quota. The only case that a new ready node is added in the list containing γ_{i-1} (apart from γ_i) is if γ_{i-1} is a spawn, so that the right child of γ_{i-1} becomes the ready node of the thread containing γ_{i-1}. However, this does not violate the properties. Assume now that the processor exhausted its enabling quota. Notice that in this case, the last instruction executed by the processor could not be a spawn (since the enabling occurred) and thus it is not possible that a new node is added in the list containing γ_{i-1}, apart from γ_i, so that by induction hypothesis, all claims hold.

We finally consider the case that the execution of γ_{i-1} enables γ_i. Since the thread containing γ_i was suspended, it was in a suspended list. Clearly, no other thread exist in this list by the time of the enabling, while no other thread can be inserted in this list before γ_i is enabled and gets executed. Thus, both properties hold.

If γ_i is in some list, it may either get executed on a processor or it may get stolen. If it gets executed, it must be the ready node of the bottommost thread, so that all properties hold by induction hypothesis. If it gets stolen, the ready list of the processor that performs the stealing is empty, so that the claims hold vacuously.

At this point the proof of the proposition is complete. ∎

Our next proposition states that for any instruction γ of the computation, at most $2T_\infty$ successful rounds of steal attempts can occur during all candidate timesteps $t \in \mathcal{C}_\gamma$. Similar arguments (to those presented in the proof of our proposition) have been presented in [26, Lemma 6.3, Chapter 6] for proving a more restricted version of the stated claim. This restricted claim is used in [26] only for proving the space bound of DFDeques algorithm, while our more general proposition is useful in bounding both the space and time complexity of our algorithm.

Proposition 5.7 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF*. Consider any node $\gamma \in G$ and let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ, $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ be the set of candidate intervals of γ and $\mathcal{C}_\gamma$ be the sets of candidate timesteps during $\mathcal{I}_\gamma$. Then, the total number of successful rounds of steal attempts in $\mathcal{C}_\gamma$ is at most $2T_\infty$.*

Proof: Let $C_i \in \mathcal{C}_\gamma$ be any set of candidate timesteps. Clearly, for all timesteps $t \in C_i$, t is a candidate timestep for some ready node $\gamma_i \in \mathcal{P}_\gamma$. Let n_i be the number of threads above γ_i in γ_i's ready list. Clearly, γ_i must get executed before or by the time that $n_i + 1$ successful steal attempts target γ_i's list. Thus, there can be at most $n_i + 1$ successful rounds in C_i, so that the total number of successful rounds in $\mathcal{C}_\gamma$ is at most

$$\begin{aligned} \sum_{i=1}^{\delta(\gamma)} (n_i + 1) &= \delta(\gamma) + \sum_{i=1}^{\delta(\gamma)} n_i \\ &\leq T_\infty + \sum_{i=1}^{\delta(\gamma)} n_i \,. \end{aligned}$$

Lemma 5.6 implies that all n_i ready nodes above γ_i in its ready list are right children of nodes in $\mathcal{P}_\gamma$. Since $\mathcal{P}_\gamma$ is a path, there can be at most T_∞ nodes in $\mathcal{P}_\gamma$ and each one can have at most one right child. Thus,

$$\sum_{i=1}^{\delta(\gamma)} n_i \leq T_\infty \,,$$

so that the total number of successful rounds in $\mathcal{C}_\gamma$ is at most $2T_\infty$, as needed. ∎

Our next proposition states that if a node γ is the last non-premature node in some parallel prefix $\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$, then the total number of rounds in $\mathcal{C}_\gamma$ plus the total number of rounds in the interval following the execution of γ till timestep t is bounded from above.

Proposition 5.8 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. Consider any execution schedule $\mathcal{X}_{\mathsf{PSDF}}(G,P)$ and any integer $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|\,]$. Let γ be the last node in $(\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$ and let t_γ be the timestep that γ is executed. Assume that $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of γ, $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ is the set of critical intervals of γ, and $\mathcal{C}_\gamma$ is the set of sets of candidate timesteps during $\mathcal{I}_\gamma$. If $I'_\gamma = \{t_\gamma, \ldots, t\}$, then,*

(1) *the number of successful rounds in $\mathcal{C}_\gamma \cup I'_\gamma$ is $O(T_\infty)$;*

(2) *the expected number of rounds in $\mathcal{C}_\gamma \cup I'_\gamma$ is $O(T_\infty)$;*

(3) *for any $\varepsilon > 0$, the number of rounds in $\mathcal{C}_\gamma \cup I'_\gamma$ is $O(T_\infty + \ln(1/\varepsilon))$, with probability at least $1 - \varepsilon$.*

Proof: Part of our proof follows similar ideas as those presented in [26, Lemma 6.3, Chapter 6]. The ideas presented there can also be applied here because our algorithm maintains important properties, like those proved in Propositions 5.3 and 5.6.

Lemma 5.7 implies that the total number of successful rounds in $\mathcal{C}_\gamma$ is $O(T_\infty)$. We now prove that at most $O(T_\infty)$ successful rounds occur in I'_γ. Let γ' be the ready node at the start of interval I'_γ with the highest priority. Clearly, $\gamma' \notin (\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{\mathsf{DF}}(G)$, because otherwise γ' and not γ would have been the last non-premature node to be executed in $(\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{\mathsf{DF}}(G)$. Notice that $\gamma' \notin \mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$ because otherwise it would be also the case that $\gamma' \in (\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{\mathsf{DF}}(G)$. If γ' is about to be executed on a processor, then I'_γ is empty. Otherwise, since γ' is the ready node of highest priority, it resides in the globally leftmost ready list. Lemma 5.5 implies that all ready nodes of threads above γ' in its ready list are right children of nodes in $\mathcal{P}_{\gamma'}$. Clearly, there should be at most T_∞ such nodes. Thus, I'_γ can contain at most T_∞ successful rounds, so that there exist $O(T_\infty)$ successful rounds in $\mathcal{C}_\gamma \cup I'_\gamma$, as needed by (1).

Lemma 3.1 implies that each round succeeds with constant probability, so that the expected number of rounds is $O(T_\infty)$, as needed by (2).

Let X be the random variable denoting the total number of successful rounds in $\mathcal{C}_\gamma \cup I'_\gamma$. Recall that the total number of successful rounds is $O(T_\infty)$, so that there exists a constant $c > 0$ such that $X < cT_\infty$. Assume that there are more than $4cT_\infty + 8\ln(1/\varepsilon)$ rounds. Then, the expected number of successful rounds is at least $\mu = 2cT_\infty + 4\ln(1/\varepsilon)$. Let $\delta = (cT_\infty + 4\ln(1/\varepsilon))/\mu$. Clearly, $\delta < 0$, while $\mu\delta = cT_\infty + 4\ln(1/\varepsilon)$ and $\mu - \mu\delta = cT_\infty$. By applying the Chernoff bound [24, Chapter 4, Section 4.1,Theorem 4.2] we have[9]:

$$\mathbf{Pr}[X < \mu - \mu\delta] \quad < \quad \exp\left\{-\frac{(\mu\delta)^2}{2\mu}\right\},$$

[9]The probability of success of a round is not necessarily independent of previous rounds, but we can apply the Chernoff bound, because each round succeeds with probability at least 1/2, independently of other rounds.

so that

$$\begin{aligned}
\mathbf{Pr}[X < cT_\infty] &< \exp\left\{-\frac{(cT_\infty + 4\ln(1/\varepsilon))^2}{2\,(2cT_\infty + 4\ln(1/\varepsilon)}\right\} \\
&= \exp\left\{-\frac{(cT_\infty + 4\ln(1/\varepsilon))^2}{4\,(cT_\infty + 4\ln(1/\varepsilon)) - 8\ln(1/\varepsilon)}\right\} \\
&\le \exp\left\{-\frac{(cT_\infty + 4\ln(1/\varepsilon))^2}{4\,(cT_\infty + 4\ln(1/\varepsilon))}\right\} \\
&= \exp\left\{-\frac{cT_\infty + 4\ln(1/\varepsilon)}{4}\right\} \\
&\le \exp\left\{-\frac{4\ln(1/\varepsilon)}{4}\right\} \\
&= e^{\ln(1/\varepsilon)} \\
&= \varepsilon\,.
\end{aligned}$$

Because there can be at most cT_∞ successful rounds, the number of required rounds is more than $4cT_\infty + 8\ln(1/\varepsilon)$ rounds with probability at most ε. Thus, we conclude that the total number of rounds in $\mathcal{C}_\gamma \cup I'_\gamma$ is $O(T_\infty + \ln(1/\varepsilon))$, with probability at least $1-\varepsilon$, as needed by (3). ■

The next proposition provides bounds for the number of heavy premature nodes in any parallel prefix of $\mathcal{X}_{\mathsf{PSDF}}(G,P)$.

Proposition 5.9 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. Consider any execution schedule $\mathcal{X}_{\mathsf{PSDF}}(G,P)$ and any integer $t \in |\mathcal{X}_{\mathsf{PSDF}}(G,P)|$. Let γ be the last node in $(\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$ and let t_γ be the timestep that γ is executed. Assume that $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of γ, $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ is the set of critical intervals of γ, $\mathcal{C}_\gamma$ is the set of sets of candidate timesteps during $\mathcal{I}_\gamma$, and $I'_\gamma = \{t_\gamma, \ldots, t\}$. If R_t is the number of rounds of steal attempts in $\mathcal{C}_\gamma \cup I'_\gamma$, then the number of heavy premature nodes in $\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$ is $O(PT_\infty + PR_t)$.*

Proof: Clearly, for each $i \in [\delta(\gamma)]$, $\gamma_i \in \mathcal{P}_\gamma$ is an ancestor of γ. Since γ is non-premature, γ_i is also non-premature. Clearly, $t_\gamma \le t$. Thus, at each timestep t', there exists some node $\gamma_i \in \mathcal{P}_\gamma$ such that γ_i is ready at t', so that γ_i is in some ready list. If the list containing γ_i has been inserted in the working tree by timestep t', Lemma 3.2 implies that t' is a candidate timestep for heavy premature nodes only if it is also a candidate timestep for γ_i. Thus, a heavy premature node can be stolen and executed only at some timestep t' such that:

(1) $t' \in \mathcal{I}_\gamma$, but $t' \notin \mathcal{C}_\gamma$; that is, the ready node $\gamma_i \in \mathcal{P}_\gamma$ at t' is in some ready list which has not yet been inserted in the working tree (and thus steal attempts can not target this ready list). Call this sort of timesteps, timesteps of type A;

(2) $t' \in \mathcal{C}_\gamma \cup \mathcal{I}'_\gamma$; call this sort of timesteps, timesteps of type B.

We first derive an upper bound on the number of heavy premature nodes that can be stolen and executed at timesteps of type A. Clearly, for any timestep $t' \in \mathcal{I}_\gamma$, only one of the nodes $\gamma_i \in \mathcal{P}_\gamma$ is ready in some ready list. Due to the way insertion of newly created lists (residing in the steal array) to the working tree is performed, a ready list is not an element of the working tree for the duration of at most one scheduling iteration. Clearly, $\delta(\gamma) < T_\infty$, so that there exist at most T_∞ different ready lists in which each one of the $\gamma_i \in \mathcal{P}_\gamma$ resides. Since only one steal attempt can be successful in each of the $(1-\alpha)P$-highest priority ready lists at each scheduling iteration, at most $(1-\alpha)P$ heavy premature nodes can be stolen and executed at each scheduling iteration. All in all, all timesteps of type A cover the duration of at most T_∞ scheduling iterations and for each such scheduling iteration at most P heavy premature nodes can be stolen and executed. Thus, the total number of heavy premature nodes executed at timesteps of type A is at most PT_∞.

We now concentrate on timesteps of type B. By assumption it holds that the total number of rounds in $\mathcal{C}_\gamma \cup I'_\gamma$ is R_t. Each round may result in $O(P)$ heavy premature nodes to be stolen and executed. Moreover, in each interval I_i another $P-1$ heavy premature nodes may be executed in the same timestep that γ_i is executed. Clearly, $\mathcal{P}_\gamma$ contains at most T_∞ nodes. Thus, the total number of heavy premature nodes executed at timesteps of type B is $O(PT_\infty + PR_t)$.

Clearly, the total number of heavy premature nodes at t is the sum of the heavy premature nodes executed at timesteps of type A and at timesteps of type B. Thus, the total number of heavy premature nodes at t is $O(PT_\infty + PR_t)$, as needed. ∎

The following corollary is an immediate consequence of Propositions 5.8 and 5.9.

Corollary 5.10 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. Consider any execution schedule $\mathcal{X}_{\mathsf{PSDF}}(G,P)$ and any integer $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|]$. Then,*

(1) *the expected number of heavy premature nodes in $\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$ is $O(PT_\infty)$;*

(2) *for any $\varepsilon > 0$, the number of heavy premature nodes in $\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$ is $O(P(T_\infty + \ln(1/\varepsilon))$ with probability at least $1-\varepsilon$.*

The next proposition bounds the size of the working tree.

Proposition 5.11 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G,P)$ is any execution schedule produced by algorithm* PSDF*, let t be any integer such that $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|]$, 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, $|\mathcal{B}_t| \le 2(H_t + T_\infty)$.*

Proof: All lists which exist in the system at t has been created either by stealing or by suspension of some thread. We first concentrate on the number of lists that have been created by stealing.

Let L_1 be any list created by stealing, let $t_1 < t$ be the timestep at which L_1 was created, and let p_1 be the processor that performed the stealing. Since a successful steal attempt causes the creation of a new list, the stolen node, let it be γ_1, must be either a non-premature node or a heavy premature node at t_1. If γ_1 is a non-premature node at t_1, it is also a non-premature node at t. If γ_1 is a heavy premature node at t_1, it can be either a non-premature node or a heavy premature node at t. We proceed by case analysis on γ_1.

Assume first that γ_1 is a non-premature node at t. All nodes executed by p_1 from timestep t_1 till its next steal attempt form a contiguous subsequence of the DF-schedule. Thus, all the executed nodes of threads contained in L_1 are non-premature. Thus, there is no thread in L_1 containing some premature node at t. Recall that all non-premature nodes belong to a serial execution. Clearly, the maximum number of active threads in a serial execution cannot be more than T_∞, so that there exist at most T_∞ active threads at t containing only non-premature nodes. Thus, there exist at most T_∞ ready lists at t created in this way.

Assume now that γ_1 is a heavy premature node at t. Clearly, for each heavy premature node at t there can be at most one list which has been created by stealing and execution of this node. Thus, there exists a one-to-one correspondence between the heavy premature nodes and this sort of lists. Thus, the number of lists of this type at t is at most H_t, so that the number of lists at t, which has been created by stealing is at most $H_t + T_\infty$.

We now bound the number of lists which have been created by suspension of threads. Consider any such list L_2. List L_2 contains only one thread which has not been worked on since it stalled. This is so because the only way for a suspended thread to be worked on is if it is enabled and some processor steals it from the suspended list, thus, putting it into a list which has been created by stealing (this case has already been studied above). Let $t_2 < t$ be the timestep at which L_2 was created. At timestep t_2 some processor p_2 was trying to execute the suspended node, let it be γ_2. Assume that Γ_2 is the thread containing γ_2, and let L_3 be the ready list, owned by p_2 at timestep t_2. Clearly, L_3 has been created by stealing. Let γ_3 be the stolen node that causes the creation of L_3, and let $t_3 < t_2$ be the timestep that the stealing occured. We proceed by case analysis on γ_3.

Assume first that γ_3 is a non-premature node at t. The nodes executed by p_2 from timestep t_3 to timestep t_2 form a contiguous subsequence of the DF-schedule. Thus, if γ_3 is non-premature, node γ_2 is either also a non-premature node at t or it is the node of globally highest priority in the system. Clearly, there exists at most one node with the globally highest priority in the system, which can reside in only one list. Moreover, if γ_2 is non-premature, all executed nodes of thread Γ_2 are non-premature nodes at t. Recall that all non-premature nodes belong to a serial execution. Clearly, the maximum number of active threads in a serial execution cannot be more than T_∞, so that there exist at most T_∞ active threads at t containing only non-premature nodes. Thus, there exist at most T_∞ lists at t, created in this way.

We now assume that γ_3 is a heavy premature node at t. Recall that for each heavy premature node there exists only one list which has been created by stealing and executing this node. Moreover, only one thread of the list can suspend, so that for each such list there exists at most one additional suspended list. Thus, there exists a one-to-one correspondence between the heavy premature nodes at t and this sort of lists. Thus, the total number of lists at t created by suspension is at most $H_t + T_\infty$.

Since, at timestep t, the total number of lists created by stealing is at most $H_t + T_\infty$, and the number of lists created by suspension of threads is at most $H_t + T_\infty$, we conclude that the total number of lists at t is at most $2(H_t + T_\infty)$; that is, $|\mathcal{B}_t| \leq 2(H_t + T_\infty)$, as needed. ■

Corollary 5.10 and Proposition 5.11 immediately imply:

Corollary 5.12 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF, *let t be any integer such that $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|]$, and $|\mathcal{B}_t|$ be the size of the working tree at t. Then,*

(1) *the expected number of lists of threads in the system at t is $O(PT_\infty)$, so that the expected size of the working tree at t is $O(PT_\infty)$; that is, $\mathcal{E}(|\mathcal{B}_t|) \in O(PT_\infty)$;*

(2) *for any $\varepsilon > 0$, the total number of lists of threads in the system at t is $O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$, so that the size of the working tree at t is $O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$; that is, $|\mathcal{B}_t| \in O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$.*

5.4 Space Bound

Let $S^D_{\mathsf{PSDF}}(G, P)$ be the random variable denoting the memory used by our algorithm to store all the data structures it maintains. Our first proposition provides bounds for S^D_{PSDF}.

Proposition 5.13 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF*, let t be any integer such that $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G, P)|]$, and $|\mathcal{B}_t|$ be the size of the working tree at t. Then,*

(1) $\mathcal{E}(S^D_{\mathsf{PSDF}}(G, P)) \in O(PT_\infty)$;

(2) *for any $\varepsilon > 0$, $S^D_{\mathsf{PSDF}}(G, P) \in O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$.*

Proof: We bound the space allocated for the working tree, the synchronization queues and the working arrays. Since the working-tree is a 2-3 tree (that is, a balanced tree in which each internal node has 2 or 3 children), the memory spent for storing the whole tree is $O(|\mathcal{B}_t|)$, where $|\mathcal{B}_t|$ is the number of the leaves of the tree.

Clearly, suspended threads in the system are stored either in a separate list of the working tree or in the stall array. Thus, the total number of suspended threads in the system is $O(P + |\mathcal{B}_t|)$. Therefore, the total number of elements in the synchronization queues is $O(P + |\mathcal{B}_t|)$. Since there are $O(P + |\mathcal{B}_t|)$ suspended threads, there exist at most $O(P + |\mathcal{B}_t|)$ synchronization variables with waiting threads, so that the total size of the enabling array E and all lists indexed by its elements is $O(P + |\mathcal{B}_t|)$[10]. For each of the other working arrays, the memory allocated is $O(P)$. Thus, Corollary 5.12 implies the stated claims. ■

We continue to bound the space complexity of our algorithm. Clearly, the space complexity of any depth-first scheduling algorithm depends on the memory threshold K. If K is fixed, Corollary 5.10 and Proposition 5.13 immediately imply:

Theorem 5.14 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. If the memory threshold K is fixed during the course of the execution of* PSDF*, it holds that:*

(1) $\mathcal{E}(S_{\mathsf{PSDF}}(G, P)) \le S_1 + O(PT_\infty K)$;

(2) *for any $\varepsilon > 0$,*

$$S_{\mathsf{PSDF}}(G, P) \quad \le \quad S_1 + O(P(T_\infty + \ln(1/\varepsilon))K)\,,$$

with probability at least $1 - \varepsilon$.

[10] We assume here that the enabling threshold $g = 1$.

Theorem 5.14 provides a bound on the space complexity of our algorithm whenever K is fixed during the course of the execution. However, we will see in Section 5.5 that the choice of K influences not only the space complexity but also the time complexity of our algorithm. It turns out that the best bound on the expected time complexity of our algorithm is achieved, whenever $K = \log(PT_\infty)$. However, it is usually the case that T_∞ is not known at the beginning of the execution. Thus, the algorithm can not always set K to $\log(PT_\infty)$ at the beginning of the execution. However, we prove in Section 5.5 that the same good time bound can be derived whenever K is variable and reset its value to $\log(|\mathcal{B}_t| + P)$ at the beginning of each scheduling iteration, where t is the first timestep of the scheduling iteration. Clearly, $|\mathcal{B}_t|$ is a random variable, so that $\log(|\mathcal{B}_t| + P)$ is also a random variable and Theorem 5.14 is not applied in this case. In what follows, we prove bounds on the space complexity under the assumption that PSDF resets K to $\log(|\mathcal{B}_t| + P)$ at the beginning of each scheduling iteration. We denote by K_t the memory threshold at timestep t[11]. We first prove:

Proposition 5.15 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF*, let t be any integer such that $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G, P)|]$, and K_t be the memory threshold at t. Then,*

(1) $\mathcal{E}(K_t) \in O(\log(PT_\infty))$*;*

(2) *for any $\varepsilon > 0$, $K_t \in O(\log(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$.*

Proof: We start by proving (1). Consider the function $\phi(x) = 2^x$. Since $\phi''(x) = 2^x > 0$, $\phi(x)$ is a convex function. Thus, by Jansen inequality,

$$\mathcal{E}(2^{K_t}) \;\geq\; 2^{\mathcal{E}(K_t)} .$$

Recall that $K_t = \log(|\mathcal{B}_t| + P)$. Thus, $2^{K_t} = 2^{\log(|\mathcal{B}_t|+P)} = |\mathcal{B}_t| + P$, so that $\mathcal{E}(2^{K_t}) = \mathcal{E}(2^{\log(|\mathcal{B}_t|+P)}) = \mathcal{E}(|\mathcal{B}_t| + P) = \mathcal{E}(|\mathcal{B}_t|) + P$. Corollary 5.12 implies that $\mathcal{E}(|\mathcal{B}_t|) \in O(PT_\infty)$. Thus, there exists some constant $c > 0$ such that $cPT_\infty + P \geq \mathcal{E}(2^{K_t}) \geq 2^{\mathcal{E}(K_t)}$, so that $2^{\mathcal{E}(K_t)} \leq cPT_\infty + P$. Since the function $\log x$ is increasing, we thus have: $\log 2^{\mathcal{E}(K_t)} \leq \log(cPT_\infty + P)$, so that $\mathcal{E}(K_t) \leq \log(cPT_\infty + P)$. Thus, $\mathcal{E}(K_t) \in O(\log(PT_\infty))$, as needed by (1).

We continue to prove (2). Corollary 5.12 implies that for any $\varepsilon > 0$, $|\mathcal{B}_t| \in O(P(T_\infty + \ln(1/\varepsilon)))$, with probability at least $1 - \varepsilon$. Thus, there exists some constant $c > 0$ such that, for any $\varepsilon > 0$, $|\mathcal{B}_t| \leq cP(T_\infty + \ln(1/\varepsilon))$, with probability at least $1 - \varepsilon$. Clearly, $|\mathcal{B}_t| + P \leq cP(T_\infty + \ln(1/\varepsilon)) + P$, with probability at least $1 - \varepsilon$. Since $\log x$ is an increasing function,

$$|\mathcal{B}_t| + P \leq cP(T_\infty + \ln(1/\varepsilon)) + P \quad \Leftrightarrow \quad \log(|\mathcal{B}_t| + P) \leq \log(cP(T_\infty + \ln(1/\varepsilon)) + P)\,,$$

so that the probability that $|\mathcal{B}_t| \in O(P(T_\infty + \ln(1/\varepsilon)))$ equals the probability that $\log(|\mathcal{B}_t| + P) \in O(\log(P(T_\infty + \ln(1/\varepsilon)) + P)$. Thus, for any $\varepsilon > 0$, the probability that $\log(|\mathcal{B}_t| + P) \in O(\log(P(T_\infty + \ln(1/\varepsilon)))$ is at least $1 - \varepsilon$, as needed by (2). ■

We continue to prove the upper bounds on the space complexity of our algorithm for the case of variable memory threshold.

[11] Clearly, for all timesteps t such that t is the first timestep of a scheduling iteration $K_t = \log(|\mathcal{B}_t| + P)$.

Theorem 5.16 *Assume that G is any multithreaded computation and let $P > 0$ be any integer. Then,*

(1) $\mathcal{E}(S_{\mathsf{PSDF}}(G,P)) \leq S_1 + O(PT_\infty \log(PT_\infty))$*;*

(2) *for any* $\varepsilon > 0$*,*

$$S_{\mathsf{PSDF}}(G,P) \quad \leq \quad S_1 + O(P(T_\infty + \ln(1/\varepsilon)) \log(P(T_\infty + \ln(P(T_\infty + \ln(1/\varepsilon))/\varepsilon)))) \,,$$

with probability at least $1 - \varepsilon$*.*

Proof: At any timestep $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|\,]$ the net memory allocated by PSDF at t is the sum of the net memory allocated for non-premature nodes and the net memory allocated for premature nodes. By definition all non-premature nodes form some prefix of $\mathcal{X}_{\mathsf{DF}}(G)$ and thus their net memory allocation is at most S_1.

Let S_p be the random variable denoting the net memory allocated by the premature nodes. Clearly, each premature node belongs to a batch of nodes, for which there exists a heavy premature node at t. Recall that we have assumed that it is the heavy node of a batch that allocates the memory needed by all nodes in the batch, so that the amount of memory allocated by premature nodes is the amount of memory allocated by heavy premature nodes. 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 has been executed. Then, the node can not allocate more than $K_{t_i} = \log(|\mathcal{B}_{t_i}| + P)$ memory. Thus, the total net memory allocated at t is:

$$S_p \quad = \quad \sum_{i=1}^{H_t} \log(|\mathcal{B}_{t_i}| + P) \,.$$

We want to study $\mathcal{E}(S_p)$. We define the random variable $S_p^{H_t}$ to be $S_p^{H_t} = \mathcal{E}(S_p \mid H_t)$, so that

$$\begin{aligned}
S_p^{H_t}(n) \quad &= \quad \mathcal{E}(S_p \mid H_t = n) \\
&= \quad \mathcal{E}(\sum_{i=1}^{n} \log(|\mathcal{B}_{t_i}| + P)) \\
&= \quad \sum_{i=1}^{n} \mathcal{E}(\log(|\mathcal{B}_{t_i}| + P)) \\
& \qquad \text{(by linearity of expectation)} \,.
\end{aligned}$$

Proposition 5.15 implies that for each $t_i \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|]$, $\mathcal{E}(\log(|\mathcal{B}_{t_i}| + P)) \in O(\log(PT_\infty))$. Thus, there exists some constant $c > 0$ such that:

$$\begin{aligned}
S_p^{H_t}(n) \quad &\leq \quad c \sum_{i=1}^{n} \log(PT_\infty) \\
&= \quad cn \log(PT_\infty) \,.
\end{aligned}$$

Clearly, $\mathcal{E}(S_p^{H_t}) = \mathcal{E}(\mathcal{E}(S_p \mid H_t))$, while Lemma 2.1 implies that $\mathcal{E}(S_p) = \mathcal{E}(\mathcal{E}(S_p \mid H_t))$. Thus,

$$\mathcal{E}(S_p) \quad = \quad \sum_{n} S_p^{H_t}(n) \, \mathbf{Pr}[H_t = n]$$

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

Corollary 5.10 implies that $\mathcal{E}(H_t) \in O(PT_\infty)$, so that $\mathcal{E}(S_p) \in O(PT_\infty \log(PT_\infty))$. By this and by Proposition 5.13(1), we conclude that $\mathcal{E}(S_{\mathsf{PSDF}}(G,P))$ is at most $S_1 + O(PT_\infty \log(PT_\infty))$, as needed by (1).

We continue to prove (2). We prove:

$$\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1\, c_2\, P(T_\infty + \ln(2/\varepsilon)) \log(P(T_\infty + \ln(2c_1 P(T_\infty + \ln(2/\varepsilon))/\varepsilon)))] \quad \geq \quad 1-\varepsilon\,,$$

where $c_1, c_2 > 0$ are constants (to be determined later).

Let γ be the last non-premature node to be executed in $\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t$; that is, γ is the last node in $(\mathcal{X}_{\mathsf{PSDF}}(G,P) \uparrow t) \mid \mathcal{X}_{DF}(G)$. Then, $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ is the critical path of γ, $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ is the set of critical intervals of γ and $\mathcal{C}_\gamma = \{C_1, \ldots, C_{\delta(\gamma)}\}$ is the set of sets of candidate timesteps. Assume that t_γ is the timestep that 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 $\mathcal{C}_\gamma \cup I'_\gamma$.

For simplicity, we denote by $\alpha_\varepsilon = (T_\infty + \ln(2/\varepsilon))$. Clearly,

$$\begin{aligned}
&\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1\, c_2\, P(T_\infty + \ln(2/\varepsilon)) \log(P(T_\infty + \ln(2c_1 P(T_\infty + \ln(2/\varepsilon))/\varepsilon)))] \\
=\;& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\, \alpha_\varepsilon\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))] \\
\geq\;& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\, \alpha_\varepsilon\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon)) \cap R_t \leq c'\, \alpha_\varepsilon] \\
&\text{(where } c' > 0 \text{ is some constant (to be determined later))} \\
=\;& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\, \alpha_\varepsilon\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon)) \mid R_t \leq c'\, \alpha_\varepsilon]\, \mathbf{Pr}[R_t \leq c'\, \alpha_\varepsilon] \\
&\text{(by the definition of conditional probability)}\,.
\end{aligned}$$

Proposition 5.8 implies that there exists some constant $c > 0$ such that $\mathbf{Pr}[R_t \leq c\,(T_\infty + \ln(2/\varepsilon))] \geq 1 - \varepsilon/2$. Let c' be this constant (that is, $c' = c$), so that

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

Since $R_t \leq c'\, \alpha_\varepsilon$, Proposition 5.9 implies that the number of heavy premature nodes at t cannot be more than $O(P(T_\infty + \ln(1/\varepsilon)))$. Thus, there exists some constant c'', such that $H_t \leq c'' P(T_\infty + \ln(1/\varepsilon))$. Let $c_1 = c''$, so that $H_t \leq c_1 P\, \alpha_\varepsilon$. Since $H_t \leq c_1 P\, \alpha_\varepsilon$, it holds that $\sum_{i=1}^{H_t} K_{t_i} \leq \sum_{i=1}^{c_1 P \alpha_\varepsilon} K_{t_i}$, where we set $K_{t_i} = K_t$, for all $i = H_t, \ldots, c_1 P \alpha_\varepsilon$. Thus, if $\sum_{i=1}^{c_1 P \alpha_\varepsilon} K_{t_i} \leq c_1 c_2 P\, \alpha_\varepsilon \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))$, then it also holds that $\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P\, \alpha_\varepsilon \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))$, so that

$$
\begin{aligned}
& \mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1 c_2 P(T_\infty + \ln(2/\varepsilon)) \log(P(T_\infty + \ln(2c_1 P(T_\infty + \ln(2/\varepsilon))/\varepsilon))] \\
\geq\ & \mathbf{Pr}[\sum_{i=1}^{c_1 P \alpha_\varepsilon} K_{t_i} \leq (c_1 P\, \alpha_\varepsilon)\, c_2 \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon)) \mid R_t \leq c'\, \alpha_\varepsilon]\, (1 - \varepsilon/2) \\
\geq\ & \mathbf{Pr}[(\bigcap_{i=1}^{c_1 P \alpha_\varepsilon} K_{t_i} \leq c_2\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\, \alpha_\varepsilon]\, (1 - \varepsilon/2) \\
=\ & \mathbf{Pr}[\bigcap_{i=1}^{c_1 P \alpha_\varepsilon} (K_{t_i} \leq c_2\, \log(P(T_\infty + \ln(2c_1 \alpha_\varepsilon/\varepsilon)) \mid R_t \leq c'\, \alpha_\varepsilon)]\, (1 - \varepsilon/2) \\
=\ & (1 - \mathbf{Pr}[(\bigcup_{i=1}^{c_1 P \alpha_\varepsilon} (K_{t_i} > c_2\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon)) \mid R_t \leq c'\, \alpha_\varepsilon)])\, (1 - \varepsilon/2) \\
\geq\ & (1 - \sum_{i=1}^{c_1 P \alpha_\varepsilon} \mathbf{Pr}[K_{t_i} > c_2\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\, \alpha_\varepsilon])\, (1 - \varepsilon/2)\,.
\end{aligned}
$$

Consider any arbitrary $i \in [c_1 P \alpha_\varepsilon]$. We prove the following lemma.

Lemma 5.17 *It holds that:*

$$\mathbf{Pr}[K_{t_i} > c_2\, \log(P(T_\infty + \ln(2c_1 P\, \alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\, (T_\infty + \ln(2/\varepsilon))] \quad < \quad \varepsilon/(2c_1 P \alpha_\varepsilon)\,.$$

Proof: Let γ' be the last non-premature node to be executed in $\mathcal{X}_{\mathsf{PSDF}}(G, P) \uparrow t_i$; that is, γ' is the last node in $(\mathcal{X}_{\mathsf{PSDF}}(G, P) \uparrow t_i) \mid \mathcal{X}_{DF}(G)$. Then, $\mathcal{P}_{\gamma'} = \langle \gamma'_1, \ldots, \gamma'_{\delta(\gamma')} \rangle$ is the critical path of γ', $\mathcal{I}_{\gamma'} = \{I'_1, \ldots, I'_{\delta(\gamma')}\}$ is the set of critical intervals of γ' and $\mathcal{C}_{\gamma'} = \{C'_1, \ldots, C'_{\delta(\gamma')}\}$ is the set of sets of candidate timesteps. Assume that $t_{\gamma'}$ is the timestep that γ' is executed and let $I'_{\gamma'} = \{t_{\gamma'}, \ldots, t_i\}$.

We partition all timesteps $t \in \mathcal{C}_{\gamma'} \cup I'_{\gamma'}$ into timesteps of type K, and timesteps of type L. A timestep t is a timestep of type K if $t \in (\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \cap (\mathcal{C}_\gamma \cup I'_\gamma)$, while it is a timestep of type L if $t \in (\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \setminus ((\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \cap (\mathcal{C}_\gamma \cup I'_\gamma))$.

The fact that $R_t \leq c'\, (T_\infty + \ln(2/\varepsilon))$ means that the total number of rounds that occur in $\mathcal{C}_\gamma \cup I'_\gamma$ is at most $c'\, (T_\infty + \ln(2/\varepsilon))$, so that the total number of rounds that occur in $(\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \cap (\mathcal{C}_\gamma \cup I'_\gamma)$ is at most $c'\, (T_\infty + \ln(2/\varepsilon))$. Thus, at most $c'\, (T_\infty + \ln(2/\varepsilon))$ rounds occur at timesteps of type K.

We now concentrate on timesteps of type L. Clearly, rounds occuring in timesteps of type L do not share steal attempts with rounds occuring in timesteps of type K. Moreover, each round of type L contains at least P steal attempts which choose *uniformly at random* one of the $(1 - \alpha)P$ ready lists, among which there always exists a particular list of interest. On the other hand, the information $R_t \leq c'\, (T_\infty + \ln(2/\varepsilon))$ concerns only rounds in timestemps of type K. We conclude that the fact $R_t \leq c'\, (T_\infty + \ln(2/\varepsilon))$ cannot influence the probability of success of a round in timesteps of type L. Thus, given that $R_t \leq c'\, (T_\infty + \ln(2/\varepsilon))$, each round of type L succeeds with probability at least $1-(1-1/P)^P \geq 1-1/e > 1/2$, independently of other rounds of type L. Moreover, Proposition 5.8(1) implies that at most $O(T_\infty)$ succesful rounds can occur in $\mathcal{C}_{\gamma'} \cup I'_{\gamma'}$, so that at most $O(T_\infty)$ successful rounds can occur in $(\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \setminus ((\mathcal{C}_{\gamma'} \cup I'_{\gamma'}) \cap (\mathcal{C}_{\gamma} \cup I'_{\gamma}))$. Let R_L be the total number of rounds in timesteps of type L. Since there exist $O(T_\infty)$ successful rounds in timesteps of type L and $\mathbf{Pr}[\text{a round of type } L \text{ is successful} \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] > 1/2$, independently of other rounds of type L, we have that $\mathcal{E}(R_L \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))) \in O(T_\infty)$. Thus, by applying the Chernoff bound [24, Chapter 4, Section 4.1,Theorem 4.2] (in exactly the same way, as we did in the proof of Proposition 5.8) for $\varepsilon' = \varepsilon/(2c_1P\alpha_\varepsilon)$, we have that there exists some constant c such that $\mathbf{Pr}[R_L > 4cT_\infty + 8\ln(1/\varepsilon') \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] < \varepsilon'$. Clearly, $\varepsilon' < \varepsilon$, so that $c'\,(T_\infty + \ln(2/\varepsilon)) \in O(T_\infty + \ln(1/\varepsilon'))$. Since the total number of rounds of type K is at most $c'\,(T_\infty + \ln(2/\varepsilon)) \in O(T_\infty + \ln(1/\varepsilon'))$, we conclude that if R_{t_i} is the total number of rounds in $\mathcal{C}_{\gamma'} \cup I'_{\gamma'}$, then $\mathbf{Pr}[R_{t_i} \in O(T_\infty + \ln(1/\varepsilon')) \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] \geq 1-\varepsilon'$. Thus, if H_{t_i} is the number of heavy premature nodes at t_i, Proposition 5.9 implies that $\mathbf{Pr}[H_{t_i} \in O(P(T_\infty + \ln(1/\varepsilon'))) \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] \geq 1-\varepsilon'$. This and Proposition 5.11 imply that, if $|\mathcal{B}_{t_i}|$ is the size of the working tree at t_i, then it holds that $\mathbf{Pr}[|\mathcal{B}_{t_i}| \in O(P(T_\infty + \ln(1/\varepsilon'))) \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] \geq 1-\varepsilon'$. Since $\log x$ is an increasing function, for any constant $c > 0$, it holds that

$$|\mathcal{B}_{t_i}| + P \leq cP(T_\infty + \ln(1/\varepsilon)) + P \quad \Leftrightarrow \quad \log(|\mathcal{B}_t| + P) \leq \log(cP(T_\infty + \ln(1/\varepsilon)) + P)\,,$$

so that $\mathbf{Pr}[\log(|\mathcal{B}_{t_i}| + P) \in O(\log(P(T_\infty + \ln(1/\varepsilon')))) \mid R_t \leq c'\,(T_\infty + \ln(2/\varepsilon))] \geq 1-\varepsilon'$. Recall that $K_{t_i} = \log(|\mathcal{B}_{t_i}| + P)$, $\varepsilon' = \varepsilon/(2c_1P\alpha_\varepsilon)$, while $\alpha_\varepsilon = T_\infty + \ln(2/\varepsilon)$, so that there exists some constant, let it be c_2, such that $\mathbf{Pr}[K_{t_i} > c_2\,\log(P(T_\infty + \ln(2c_1P\,\alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\,\alpha_\varepsilon] < \varepsilon/(2c_1P\alpha_\varepsilon)$, as needed. ∎

Recall that

$$\begin{aligned}
&\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1c_2P(T_\infty + \ln(2/\varepsilon))\log(P(T_\infty + \ln(2c_1P(T_\infty + \ln(2/\varepsilon))/\varepsilon))] \\
\geq\;& (1 - \sum_{i=1}^{c_1P\alpha_\varepsilon} \mathbf{Pr}[K_{t_i} > c_2\,\log(P(T_\infty + \ln(2c_1P\,\alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\,\alpha_\varepsilon])\,(1-\varepsilon/2)\,.
\end{aligned}$$

Moreover, Lemma 5.17 implies that $\mathbf{Pr}[K_{t_i} > c_2\,\log(P(T_\infty + \ln(2c_1P\,\alpha_\varepsilon/\varepsilon))) \mid R_t \leq c'\,\alpha_\varepsilon] \leq \varepsilon/(2c_1P\,\alpha_\varepsilon)$. Thus,

$$\begin{aligned}
&\mathbf{Pr}[\sum_{i=1}^{H_t} K_{t_i} \leq c_1\,c_2\,P(T_\infty + \ln(2/\varepsilon))\log(P(T_\infty + \ln(2c_1P(T_\infty + \ln(2/\varepsilon))/\varepsilon)))] \\
\geq\;& (1 - \frac{c_1P\alpha_\varepsilon\,\varepsilon}{2c_1P\alpha_\varepsilon})\,(1-\varepsilon/2) \\
=\;& (1-\varepsilon/2)\,(1-\varepsilon/2) \\
=\;& 1 - \varepsilon + \varepsilon^2/4 \\
\geq\;& 1-\varepsilon\,.
\end{aligned}$$

This and Proposition 5.13(2) implies that claim (2) holds.

At this point the proof of the theorem is complete. ∎

5.5 Time Bound

We first bound the execution time of a scheduling iteration and we then prove bounds for the expected execution time of our algorithm. Blelloch *et al.* have proved in [6] the following proposition for a 2-3 priority tree.

Proposition 5.18 (Blelloch, Gibbons, Matias, Narlikar) *Using a 2-3 tree of size n, each of the priority tree operations on m elements can be performed in $O(m \log n/P)$ time on a P-processor EREW PRAM.*

Our next proposition bounds the execution time of any scheduling iteration.

Proposition 5.19 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G, P)$ is any execution schedule produced by algorithm* PSDF. *Consider any scheduling iteration i during the execution that lasts for T_i timesteps, let E_i be the random variable denoting the number of threads that should be enabled in the iteration, and let $|\mathcal{B}_i|$ be the random variable denoting the size of the working tree at the beginning of the iteration. Then, it holds that*

(1) $T_i \geq 3\log(|\mathcal{B}_i| + P)$;

(2) $T_i \in O((E_i/P + 1)\log(|\mathcal{B}_i| + P))$.

Proof: We start by proving (1). Recall that if the scheduling processors finish the five tasks of a scheduling iteration in time less than $3\log(|\mathcal{B}_i| + P)$, they wait till $3\log(|\mathcal{B}_i| + P)$ timesteps have elapsed, in order to give the opportunity to the working processors to perform a satisfactory number of steal attempts. Thus, claim (1) is trivially true.

We continue to prove (2). During any scheduling iteration the scheduling processors perform the following scheduling tasks: (1) splitting task, (2) list suspension task, (3) deletion task, (4) enabling task, and (5) activation task. We bound the total number of timesteps required to execute each of these scheduling tasks separately.

We start by bounding the execution time of the splitting task. The scheduling processors access all elements of array Σ and for each of its $O(P)$ elements, they split the corresponding element of the working tree (to which the element of Σ keeps a pointer) into two elements. Clearly, the size of the working tree is $|\mathcal{B}_i|$ when the splitting task starts being executed, while array Σ contains $2(1-\alpha)P$ elements. Since the working tree is implemented as a black-white priority tree, Proposition 5.18 implies that the time to execute the splitting task by αP processors is $O(2(1-\alpha)P \log|\mathcal{B}_i|/(\alpha P)) = O(\log|\mathcal{B}_i|)$. After the splitting task has executed, the size of the working tree may have been increased to $|\mathcal{B}_i| + O(P)$.

We proceed to bound the execution time of the list suspension task. During the execution of this scheduling task at most $O(P)$ elements of the working tree have to be split into two elements, one of which should be colored black and the other white. By Proposition 5.18, this is executed in $O(\log(|\mathcal{B}_i| + P))$ timesteps by $O(P)$ processors. Clearly, the list suspension task may increase the size of the working tree by an additive factor of $O(P)$.

During the deletion task the scheduling processors may have to delete $O(P)$ elements from the working tree. By Proposition 5.18, this is executed in $O(\log(|\mathcal{B}_i| + P))$ timesteps. After the deletion task, the size of the working tree is $O(|\mathcal{B}_i| + P)$, in the worst case.

During the enabling task the following happens. The scheduling processors first perform a prefix sums computation on $3(1-\alpha)P$ elements. This can be performed in $O(\log P)$ time by $O(P)$ processors (see, e.g. [21, Section 2.1, Chapter 2]). Each processor spends another $O(\log P)$ timesteps to execute the BinarySearch procedure on the sorted array of the prefix sums. The scheduling processors should then process each one of the synchronization variables contained in the lists of the array E. Since all elements in each such list are allocated by a contiguous chunk of memory and by Lemma 5.4 there is a total of $O(P^{g+1})$ elements in all lists, their processing time is $O(P^{g+1}/P)$. However, $P^{g+1} \leq E_i$, so that this processing time is $O(E_i/P)$. The scheduling processor perform next another prefix sums computation on $O(P^{g+1})$ elements, which can be performed in $O(\log P^{g+1}) = O(\log P)$ time (since g is a constant) by $O(P)$ processors. Each scheduling processor pays also $O(\log P)$ for performing the new BinarySearch computations. Finally, the processors access all enabled threads in the synchronization queues and for each one of them the corresponding lists in the working tree should be recolored from white to black. Proposition 5.18 implies that this can be done in $O(E_i \log(|\mathcal{B}_i| + P)/P)$ timesteps. Thus, the enabling task is executed in $O(\log P + E_i/P + E_i \log(|\mathcal{B}_i| + P)/P) = O((E_i/P + 1)\log(|\mathcal{B}_i| + P))$ timesteps. Clearly, the enabling task does not cause any change on the size of the working tree.

Finally, the activation task has to select m black elements of highest priority in the working tree and "move" them in the active array A. Proposition 5.18 implies that this can be done in $O(\log(|\mathcal{B}_i| + P))$ timesteps.

Summing up the time required to execute each one of the scheduling tasks, we conclude that $T_i \in O((E_i/P + 1)\log(|\mathcal{B}_i| + P))$, as needed by (2). ■

We continue to bound the time complexity of our algorithm. Recall that the time complexity of our algorithm differs for different values of the memory quota K. The best time bound is attained either if $K = \log(PT_\infty)$ or if K is reset to $\log(|\mathcal{B}_i| + P)$ at the beginning of any scheduling iteration i, where $|\mathcal{B}_i|$ is the size of the working tree at the beginning of the iteration. Clearly, $\log(|\mathcal{B}_i| + P)$ is a random variable, fact which significantly complicates our analysis. However, since it is usually the case that T_∞ is not known at the beginning of the execution, the proof presented below considers this more complicated case of variable memory quota. Exactly the same arguments can be applied for the case that K is fixed, equal to $\log(PT_\infty)$, but parts of our proof would be more simplified if this were the case.

Theorem 5.20 *For any computation G and any integer $P \geq \max\{3, \log_P \log T_\infty\}$, it holds that:*

$$\mathcal{E}(T_{\mathsf{PSDF}}(G,P)) \quad \in \quad O(T_1/P + \sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))\,.$$

Proof: In order to facilitate our proof, we assume that the total number of processors is $2P$, P of which are scheduling processors and P are working processors; that is, we assume that $\alpha = 1/2$. However, we would like to point out that our asymptotic analysis holds for any number of processors (even for an odd number of processors), as well as for any α such that $0 < \alpha < 1$.

We partition the execution into epochs, where each epoch i consists of the timesteps comprising the ith scheduling iteration. For simplicity of proof, we denote by $\mathcal{B}_i$ the working set at the beginning of epoch i and let $|\mathcal{B}_i|$ be the size of $\mathcal{B}_i$. Let $\mathcal{R}_i$ be the set of ready lists in

$\mathcal{B}_i$ and denote by $|\mathcal{R}_i|$ the size of $\mathcal{R}_i$. We also denote by E_i the random variable denoting the number of lists in the working tree containing suspended threads that are enabled during the ith scheduling iteration. Clearly, during any scheduling iteration some new lists are created due to stealing. Some of these lists, which we call *synchronization lists* of the iteration, may suspend or terminate or reach their enabling threshold before the end of the scheduling iteration. We denote by Σ_i the random variable denoting the number of synchronization lists of the ith scheduling iteration. We first prove:

Lemma 5.21 *For any random variable $N > 0$, it holds that:*

(1) $\mathcal{E}(\sum_{i=1}^{N} E_i \log(|\mathcal{B}_i| + P)) \in O(\sigma \log(PT_\infty))$;

(2) $\mathcal{E}(\sum_{i=1}^{N} \Sigma_i \log(|\mathcal{B}_i| + P)) \in O(\sigma \log(PT_\infty))$;

(3) *if* $\mathcal{E}(N) < \infty$, $\mathcal{E}(\sum_{i=1}^{N} \log(|\mathcal{B}_i| + P)) \in O(\mathcal{E}(N) \log(PT_\infty))$.

Proof: We start by proving (1). For simplicity of proof, we denote by $Y = \sum_{i=1}^{N} E_i \log(|\mathcal{B}_i| + P)$. In order to find $\mathcal{E}(Y)$, we study the random variable $\mathcal{E}(Y \mid N)$. Clearly, $\mathcal{E}(Y \mid N = n) = \mathcal{E}(\sum_{i=1}^{n} E_i \log(|\mathcal{B}_i| + P))$. Let $X(n) = \sum_{i=1}^{n} E_i \log(|\mathcal{B}_i| + P)$. We are looking for the expectation $\mathcal{E}(X)$ of the random variable X. In order to find $\mathcal{E}(X)$ we study the random variable $Z = \mathcal{E}(X \mid E_1, \ldots, E_n)$. Clearly,

$$\begin{aligned}
Z(e_1, \ldots, e_n) &= \mathcal{E}(X \mid E_1 = e_1, \ldots, E_n = e_n) \\
&= \mathcal{E}(\sum_{i=1}^{n} e_i \log(|\mathcal{B}_i| + P)) \\
&= \sum_{i=1}^{n} e_i \mathcal{E}(\log(|\mathcal{B}_i| + P)) \\
& \quad \text{(by linearity of expectation)}\,.
\end{aligned}$$

Proposition 5.15 implies that $\mathcal{E}(\log(|\mathcal{B}_i| + P)) \in O(\log(PT_\infty))$, so that there exists some constant $c > 0$ such that

$$\begin{aligned}
Z(e_1, \ldots, e_n) &\leq c \sum_{i=1}^{n} e_i \log(PT_\infty) \\
&= c \log(PT_\infty) \sum_{i=1}^{n} e_i\,.
\end{aligned}$$

Clearly, $\sum_{i=1}^{n} e_i \leq \sigma$, so that

$$Z(e_1, \ldots, e_n) \leq c\sigma \log(PT_\infty)\,.$$

It holds (see, e.g., [24, Lemma 4.9, Chapter 4]) that $\mathcal{E}(X) = \mathcal{E}(\mathcal{E}(X \mid E_1, \ldots, E_n))$, so that

$$\begin{aligned}
\mathcal{E}(X) &= \sum_{\substack{e_1, \ldots, e_n \\ e_1 + \ldots + e_n = \sigma}} \mathcal{E}(X \mid E_1 = e_1, \ldots, E_n = e_n)\, \mathbf{Pr}[e_1, \ldots, e_n] \\
&= \sum_{\substack{e_1, \ldots, e_n \\ e_1 + \ldots + e_n = \sigma}} Z(e_1, \ldots, e_n)\, \mathbf{Pr}[e_1, \ldots, e_n]
\end{aligned}$$

$$\begin{aligned}
&\le \sum_{\substack{e_1,\ldots,e_n \\ e_1+\ldots+e_n=\sigma}} c\sigma \log(PT_\infty)\,\mathbf{Pr}[e_1,\ldots,e_n] \\
&= c\sigma\log(PT_\infty) \sum_{\substack{e_1,\ldots,e_n \\ e_1+\ldots+e_n=\sigma}} \mathbf{Pr}[e_1,\ldots,e_n]\,.
\end{aligned}$$

Clearly,

$$\sum_{\substack{e_1,\ldots,e_n \\ e_1+\ldots+e_n=\sigma}} \mathbf{Pr}[e_1,\ldots,e_n] \;=\; 1\,,$$

so that $\mathcal{E}(X) \le c\sigma\log(PT_\infty)$. Recall that $\mathcal{E}(X) = \mathcal{E}(Y \mid N)$. Thus, $\mathcal{E}(Y \mid N) \le c\sigma\log(PT_\infty)$. However (see, e.g., [24, Lemma 4.9, Chapter 4]), $\mathcal{E}(Y) = \mathcal{E}(\mathcal{E}(Y \mid N))$, so that

$$\begin{aligned}
\mathcal{E}(Y) &= \sum_n \mathcal{E}(Y \mid N=n)\,\mathbf{Pr}[N=n] \\
&\le \sum_n c\sigma\log(PT_\infty)\,\mathbf{Pr}[N=n] \\
&= c\sigma\log(PT_\infty)\sum_n \mathbf{Pr}[N=n]\,.
\end{aligned}$$

Clearly, $\sum_n \mathbf{Pr}[N=n] = 1$, so that $\mathcal{E}(Y) \in O(\sigma\log(PT_\infty))$, as needed by (1).

The proof of (2) is exactly the same as the one of (1) (one should simply change E_i and e_i to Σ_i and σ_i, respectively), and thus it is omitted.

We finally prove (3). For simplicity of proof, we denote by $W = \sum_{i=1}^{N}\log(|\mathcal{B}_i| + P)$. In order to find $\mathcal{E}(W)$, we study the random variable $\mathcal{E}(W \mid N)$. Clearly,

$$\begin{aligned}
\mathcal{E}(W \mid N=n) &= \mathcal{E}(\sum_{i=1}^{n}\log(|\mathcal{B}_i| + P)) \\
&= \sum_{i=1}^{n}\mathcal{E}(\log(|\mathcal{B}_i| + P)) \\
&\quad\;\text{(by linearity of expectation)}\,.
\end{aligned}$$

Proposition 5.15 implies that $\mathcal{E}(\log(|\mathcal{B}_i| + P)) \in O(\log(PT_\infty))$, so that there exists some constant $c > 0$ such that

$$\begin{aligned}
\mathcal{E}(\sum_{i=1}^{n}\log(|\mathcal{B}_i| + P)) &\le \sum_{i=1}^{n} c\log(PT_\infty) \\
&= cn\log(PT_\infty)\,.
\end{aligned}$$

Thus, $\mathcal{E}(W \mid N=n) \le cn\log(PT_\infty)$. However (see, e.g., [24, Lemma 4.9, Chapter 4]), $\mathcal{E}(W) = \mathcal{E}(\mathcal{E}(W \mid N))$, so that

$$\begin{aligned}
\mathcal{E}(W) &= \sum_n \mathcal{E}(W \mid N=n)\,\mathbf{Pr}[N=n] \\
&\le \sum_n cn\log(PT_\infty)\,\mathbf{Pr}[N=n] \\
&= c\log(PT_\infty)\sum_n n\,\mathbf{Pr}[N=n]\,.
\end{aligned}$$

Clearly, $\sum_n n\mathbf{Pr}[N=n] = \mathcal{E}(N)$, so that $\mathcal{E}(W) \in O(\mathcal{E}(N)\log(PT_\infty))$, as needed by (3). ∎

Proposition 5.19 (1) implies that each epoch lasts at least $3\log(|\mathcal{B}_i| + P)$ timesteps. Let ρ_i be the number of steal attempts initiated the first $2\log(|\mathcal{B}_i| + P)$ timesteps of epoch i. We proceed by case analysis on ρ_i.

1. We first consider the case that $\rho_i < P\log(|\mathcal{B}_i| + P)$. Call this sort of epochs *execution-concentrated* epochs. Let T_e be the total number of timesteps spent for execution-concentrated epochs.

   Lemma 5.22 $\mathcal{E}(T_e) \in O(T_1/P + \sigma\log(PT_\infty)/P)$.

   Proof: Let N_e be the random variable denoting the number of execution-concentrated epochs for an execution schedule $\mathcal{X}_{\mathsf{PSDF}}(G,P)$. We denote by $j_1, \ldots j_{N_e}$ these epochs in $\mathcal{X}_{\mathsf{PSDF}}(G,P)$. Proposition 5.19 implies that epoch j_i needs $O((E_{j_i}/P+1)\log(|\mathcal{B}_{j_i}|+P))$ timesteps. Thus, there exists a constant $c > 0$ such that

   $$\begin{aligned} T_e &\leq c\sum_{i=1}^{N_e}((E_{j_i}/P+1)\log(|\mathcal{B}_{j_i}|+P)) \\ &= c\left(\sum_{i=1}^{N_e}\frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P) + \sum_{i=1}^{N_e}\log(|\mathcal{B}_{j_i}|+P)\right). \end{aligned}$$

   Recall that at each timestep of the execution a processor either executes an instruction or initiates a steal attempt. In $2\log(|\mathcal{B}_{j_i}| + P)$ timesteps the P processors can take $2P\log(|\mathcal{B}_{j_i}| + P)$ actions, either instruction executions or steal attempts. Since the number of steal attempts is less than $P\log(|\mathcal{B}_{j_i}| + P)$, at least $2P\log(|\mathcal{B}_{j_i}| + P) - P\log(|\mathcal{B}_{j_i}| + P) \geq P\log(|\mathcal{B}_{j_i}| + P)$ instructions are executed in the epoch. Assume that m_{j_i} is the number of instructions executed in the ith such epoch. Then, for each such i, $m_{j_i} \geq P\log(|\mathcal{B}_{j_i}| + P)$, so that $\sum_{i=1}^{N_e} m_{j_i} \geq P\sum_{i=1}^{N_e}\log(|\mathcal{B}_{j_i}| + P)$. Since the total number of instructions to be executed is T_1, it should be $\sum_{i=1}^{N_e} m_{j_i} \leq T_1$, so that $P\sum_{i=1}^{N_e}\log(|\mathcal{B}_{j_i}| + P) \leq T_1$, which implies that $\sum_{i=1}^{N_e}\log(|\mathcal{B}_{j_i}| + P) \leq T_1/P$. Therefore,

   $$T_e \leq c\left(\sum_{i=1}^{N_e}\frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P) + T_1/P\right),$$

   so that

   $$\mathcal{E}(T_e) \leq c\left(\mathcal{E}(\sum_{i=1}^{N_e}E_{j_i}\log(|\mathcal{B}_{j_i}|+P))/P + T_1/P\right).$$

   Lemma 5.21 implies that $\mathcal{E}(\sum_{i=1}^{N_e}E_{j_i}\log(|\mathcal{B}_{j_i}| + P)) \in O(\sigma\log(PT_\infty))$, so that $\mathcal{E}(T_e) \in O(T_1/P + \sigma\log(PT_\infty)/P)$, as needed. ∎

2. We now consider the case that $\rho_i \geq P\log(|\mathcal{B}_i| + P)$. Call this sort of epochs *stealing-concentrated* epochs. Assume that T_s is the random variable denoting the total number of timesteps spent for stealing-concentrated epochs.

   Recall that $\mathcal{R}_i$ is the set of ready lists in $\mathcal{B}_i$. We proceed by case analysis on $|\mathcal{R}_i|$.

a. Assume first that $|\mathcal{R}_i| < P$; that is, at the beginning of the ith epoch less than P ready lists exist in $\mathcal{B}_i$. Thus, less than P lists are available in the active array as targets for stealing. Let $T_{s_<}$ be the random variable denoting the total number of timesteps spent for stealing-concentrated epochs with less than P ready lists. Then,

Lemma 5.23 $\mathcal{E}(T_{s_<}) \in O(\sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$.

Proof: Let γ be the last executed instruction in $\mathcal{X}_{\mathsf{DF}}(G)$ (or any of them if there are more than one) and let $\mathcal{P}_\gamma = \langle \gamma_1, \ldots, \gamma_{\delta(\gamma)} \rangle$ be the critical path of γ. Assume that $\mathcal{I}_\gamma = \{I_1, \ldots, I_{\delta(\gamma)}\}$ is the set of critical intervals of γ. Clearly, $\mathcal{I}_\gamma$ covers the whole execution time. Therefore, at each timestep $t \in [\,|\mathcal{X}_{\mathsf{PSDF}}(G,P)|]$ some of the $\gamma_j \in \mathcal{P}_\gamma$, $j \in [\delta(\gamma)]$, is ready and thus it is in some ready list. Since $|\mathcal{R}_i| < P$, it holds that:

(1) either γ_j is in a ready list which has not yet been inserted in $\mathcal{B}$ (call this sort of timesteps, timesteps of type D), or

(2) t is a candidate timestep for γ_j (call this sort of timesteps, timesteps of type E).

We start by bounding the expected number of timesteps of type D. Due to the way the scheduling processors perform the insertion of newly created lists (residing in the steal array) in the working tree, a ready list is not an element of the working tree for the duration of at most one scheduling iteration. Clearly, $\delta(\gamma) \leq T_\infty$, so that there exist at most T_∞ different ready lists in which each of the $\gamma_j \in \mathcal{P}_\gamma$ resides. Thus, all timesteps of type D belong to at most T_∞ epochs. Let $j_1, \ldots, j_{T_\infty}$ be those epochs. By Proposition 5.19, each such epoch j_i, takes $O((E_{j_i}/P + 1)\log(|\mathcal{B}_{j_i}| + P))$ timesteps. Thus, if T_D is the random variable denoting the total number of timesteps of type D, there exists a constant $c > 0$ such that

$$\begin{aligned} T_D &\leq c \sum_{i=1}^{T_\infty} ((E_{j_i}/P + 1)\log(|\mathcal{B}_{j_i}| + P)) \\ &= c \sum_{i=1}^{T_\infty} \frac{E_{j_i}}{P} \log(|\mathcal{B}_{j_i}| + P) + c \sum_{i=1}^{T_\infty} \log(|\mathcal{B}_{j_i}| + P)\,, \end{aligned}$$

so that

$$\mathcal{E}(T_D) \leq c\,\mathcal{E}(\sum_{i=1}^{T_\infty} \frac{E_{j_i}}{P} \log(|\mathcal{B}_{j_i}| + P)) + c\,\mathcal{E}(\sum_{i=1}^{T_\infty} \log(|\mathcal{B}_{j_i}| + P))\,.$$

By Lemma 5.21, it follows that $\mathcal{E}(\sum_{i=1}^{T_\infty} E_{j_i} \log(|\mathcal{B}_{j_i}| + P)) \in O(\sigma \log(PT_\infty))$, and $\mathcal{E}(\sum_{i=1}^{T_\infty} \log(|\mathcal{B}_{j_i}| + P)) \in O(T_\infty \log(PT_\infty))$. Thus, $\mathcal{E}(T_D) \in O(\sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$.

We continue to bound the expected number of timesteps of type E. Recall that all timesteps of type E are candidate timesteps for some of the γ_j, $j \in [\delta(\gamma)]$. Since (1) $\rho_i \geq P \log(|\mathcal{B}_i| + P)$, (2) $|\mathcal{R}_i| < |\mathcal{B}_i|$, and (3) $|\mathcal{R}_i| < P$, we have $\rho_i \geq |\mathcal{R}_i|$. Let $\gamma' \in \mathcal{P}_\gamma$ be the ready instruction at the beginning of the ith epoch. We call the epoch successful if the ready list containing γ' receives at least one steal attempt.

Let N_E be the random variable denoting the total number of such epochs. We prove that the expected number of epochs of this type is $O(T_\infty)$; that is, $\mathcal{E}(N_E) \in O(T_\infty)$. Each epoch contains more than P steal attempts, each one of which chooses uniformly at random one of the P positions of the active array. Thus, the probability that the epoch is successful is at least $1-(1-1/P)^P \geq 1-1/e > 1/2$, independently of other epochs. Assume that Y is the random variable denoting the number of successful such epochs. We follow similar arguments as those presented in the proof of Proposition 5.7 to prove that $Y \leq 2T_\infty$. Let $C_i \in \mathcal{C}_\gamma$ be any set of candidate timesteps. Clearly, for all timesteps $t \in C_i$, t is a candidate timestep for some ready node $\gamma_i \in \mathcal{P}_\gamma$. Let n_i be the number of threads above γ_i in γ_i's ready list. Clearly, γ_i must get executed before or by the time that n_i+1 successful steal attempts target γ_i's list. Thus, there can be at most n_i+1 successful epochs in C_i, so that the total number of successful epochs in $\mathcal{C}_\gamma$ is at most

$$
\begin{aligned}
\sum_{i=1}^{\delta(\gamma)}(n_i+1) &= \delta(\gamma)+\sum_{i=1}^{\delta(\gamma)} n_i \\
&\leq T_\infty+\sum_{i=1}^{\delta(\gamma)} n_i \,.
\end{aligned}
$$

Lemma 5.6 implies that all n_i ready nodes above γ_i in its ready list are right children of nodes in $\mathcal{P}_\gamma$. Since $\mathcal{P}_\gamma$ is a path, there can be at most T_∞ nodes in $\mathcal{P}_\gamma$ and each one can have at most one right child. Thus,

$$
\sum_{i=1}^{\delta(\gamma)} n_i \leq T_\infty \,,
$$

so that the total number of successful epochs in $\mathcal{C}_\gamma$ is at most $2T_\infty$. Since the probability of an epoch to be successful is at least $1/2$, the expected number of epochs of this type is $O(T_\infty)$; that is, $\mathcal{E}(N_E) \in O(T_\infty)$.

By Proposition 5.19, each epoch j_i, takes $O((E_{j_i}/P+1)\log(|\mathcal{B}_{j_i}|+P)$ timesteps. Thus, if T_E is the random variable denoting the total number of timesteps of type E, there exists a constant $c>0$ such that

$$
\begin{aligned}
T_E &\leq c\sum_{i=1}^{N_E}((E_{j_i}/P+1)\log(|\mathcal{B}_{j_i}|+P)) \\
&= c\left(\sum_{i=1}^{N_E}\frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P)+\sum_{i=1}^{N_E}\log(|\mathcal{B}_{j_i}|+P)\right) .
\end{aligned}
$$

Thus,

$$
\mathcal{E}(T_E) \leq c\mathcal{E}(\sum_{i=1}^{N_E}\frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P))+c'\,\mathcal{E}(\sum_{i=1}^{N_E}\log(|\mathcal{B}_{j_i}|+P)) \,.
$$

Lemma 5.21 implies that (1) $\mathcal{E}(\sum_{i=1}^{N_E} E_{j_i}\log(|\mathcal{B}_{j_i}|+P)) \in O(\sigma\log(PT_\infty))$, and (2) $\mathcal{E}(\sum_{i=1}^{N_E}\log(|\mathcal{B}_{j_i}|+P)) \in O(\mathcal{E}(N_E)\log(PT_\infty))$. Recall that $\mathcal{E}(N_E) \in O(T_\infty)$. Thus, it holds that $\mathcal{E}(T_E) \in O(\sigma\log(PT_\infty)/P+T_\infty\log(PT_\infty))$.

Clearly, $T_{s<} \leq T_D + T_E$, so that $\mathcal{E}(T_{s<}) \leq \mathcal{E}(T_D) + \mathcal{E}(T_E)$. Thus, $\mathcal{E}(T_{s<}) \in O(\sigma \log(PT_\infty)/P + T_\infty \log(PT_\infty))$, as needed.

At this point the proof of Lemma 5.23 is complete. ■

b. Assume now that $|\mathcal{R}_i| \geq P$; that is, at the beginning of the ith epoch at least P lists are ready. Thus, P lists are available in the active array as targets for stealing. Let $T_{s>}$ be the random variable denoting the total number of timesteps spent for stealing-concentrated epochs with at least P ready lists. Then,

Lemma 5.24 $\mathcal{E}(T_{s>}) \in O(T_1/P + \sigma \log(PT_\infty)/P)$.

Proof: Let X_i be the random variable denoting the number of lists receiving at least one of the ρ_i steal attempts. Recall that $\rho_i \geq P\log(|\mathcal{B}_i| + P) \geq P \log P$. Let Σ_i be the random variable denoting the number of synchronization lists of the ith epoch. Then, $X_i - \Sigma_i$ of the lists that are created due to successful stealing in the ith epoch continue execution for at least $\log(|\mathcal{B}_i| + P)$ timesteps (since they continue execution till they reach their memory quota, which is $\log(|\mathcal{B}_i| + P)$). Thus, if Y_i is the random variable denoting the number of instructions executed in epoch i, it holds that $Y_i \geq (X_i - \Sigma_i)\log(|\mathcal{B}_i| + P)$. We say that epoch i is successful if $Y_i \geq ((P-1)/8 - \Sigma_i)\log(|\mathcal{B}_i| + P)$; that is, a stealing-concentrated epoch with at least P ready lists is successful if more than $((P-1)/8 - \Sigma_i)\log(|\mathcal{B}_i| + P)$ instructions are executed in the epoch. We study the probability for an epoch to be successful. Clearly,

$$\begin{aligned} & \mathbf{Pr}[Y_i \geq ((P-1)/8 - \Sigma_i)\log(|\mathcal{B}_i| + P)] \\ \geq\ & \mathbf{Pr}[(X_i - \Sigma_i)\log(|\mathcal{B}_i| + P) \geq ((P-1)/8 - \Sigma_i)\log(|\mathcal{B}_i| + P)]\,, \end{aligned}$$

while

$$\begin{aligned} & \mathbf{Pr}[(X_i - \Sigma_i)\log(|\mathcal{B}_i| + P) \geq ((P-1)/8 - \Sigma_i)\log(|\mathcal{B}_i| + P)] \\ =\ & \mathbf{Pr}[X_i > (P-1)/8]\,. \end{aligned}$$

Notice that X_i has the same distribution as a random variable denoting the number of occupied bins among P initially empty, after the throw of at least $P \log P$ balls in a uniformly at random way. Thus, by Lemma 2.5, $\mathbf{Pr}[X_i > P-1)/8] \geq 1-2e^{-(P-1)/8}$, so that an epoch is successful with probability at least $1-2e^{-(P-1)/8}$. Let $N_{s>}$ be the random variable denoting the number of *successful* stealing-concentrated epochs with more than P ready lists for some execution schedule $\mathcal{X}_{\mathsf{PSDF}}(G, P)$. Denote by $j_1, \ldots j_{N_{s>}}$ these successful epochs. We prove the following lemma.

Lemma 5.25 *It holds that:*
(1) $\sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}| + P) \leq 8T_1/(P-1) + 8\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \log(|\mathcal{B}_{j_i}| + P)/(P-1)$;
(2) $\mathcal{E}(N_{s>}) \in O(T_1/(P\log P) + \sigma \log(PT_\infty)/(P \log P))$.

Proof: Denote by m_{j_i} the number of instructions executed in the ith successful epoch. Clearly, for each $i \in [N_{s>}]$, $m_{j_i} \geq ((P-1)/8 - \Sigma_{j_i})\log(|\mathcal{B}_{j_i}| + P)$, so that $\sum_{i=1}^{N_{s>}} m_{j_i} \geq \sum_{i=1}^{N_{s>}} ((P-1)/8 - \Sigma_{j_i}) \log(|\mathcal{B}_{j_i}| + P)$. Since the total number of instructions to be executed is T_1, $\sum_{i=1}^{N_{s>}} m_{j_i} \leq T_1$, so that

$$\sum_{i=1}^{N_{s>}} ((P-1)/8 - \Sigma_{j_i}) \log(|\mathcal{B}_{j_i}| + P) \quad \leq \quad T_1 \, .$$

However,

$$\begin{aligned} & \sum_{i=1}^{N_{s>}} ((P-1)/8 - \Sigma_{j_i}) \log(|\mathcal{B}_{j_i}| + P) \\ = \quad & \frac{P-1}{8} \sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}| + P) - \sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P) \, , \end{aligned}$$

so that

$$\sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}| + P) \quad \leq \quad \frac{8T_1}{P-1} + \frac{8}{P-1} \sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P) \, ,$$

as needed by (1).

We continue to prove (2). Clearly, $\sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}| + P) \geq \sum_{i=1}^{N_{s>}} \log P = N_{s>} \log P$. This and (1) imply that

$$N_{s>} \log P \quad \leq \quad \frac{8T_1}{P-1} + \frac{8}{P-1} \sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P) \, ,$$

so that

$$N_{s>} \quad \leq \quad \frac{8T_1}{(P-1)\log P} + \frac{8}{(P-1)\log P} \sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P) \, .$$

Thus,

$$\mathcal{E}(N_{s>}) \quad \leq \quad \frac{8T_1}{(P-1)\log P} + \frac{8}{(P-1)\log P} \mathcal{E}(\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P)) \, .$$

By Lemma 5.21, $\mathcal{E}(\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P)) \in O(\sigma \log(PT_\infty))$, so that, there exists some constant $c > 0$, such that: $\mathcal{E}(\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}| + P)) \leq c\sigma \log(PT_\infty)$. Therefore, $\mathcal{E}(N_{s>}) \in O(T_1/(P \log P) + \sigma \log(PT_\infty)/(P \log P))$, as needed by (2). ■

By Proposition 5.19, epoch j_i takes $O((E_{j_i}/P+1) \log(|\mathcal{B}_{j_i}|+P))$ timesteps. Let $T_{s>}^s$ be the random variable denoting the total number of timesteps spent for *successful* stealing-concentrated epochs with more than P ready lists. Then, there exists a constant $c > 0$ such that

$$\begin{aligned} T_{s>}^s \quad & \leq \quad c \sum_{i=1}^{N_{s>}} ((E_{j_i}/P + 1) \log(|\mathcal{B}_{j_i}| + P)) \\ & = \quad c \left(\sum_{i=1}^{N_{s>}} \frac{E_{j_i}}{P} \log(|\mathcal{B}_{j_i}| + P) + \sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}| + P) \right) \, . \end{aligned}$$

Lemma 5.25 implies that

$$\sum_{i=1}^{N_{s>}} \log(|\mathcal{B}_{j_i}|+P) \quad \leq \quad \frac{8T_1}{P-1} + \frac{8}{P-1}\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}|+P)\,,$$

so that

$$T^s_{s>} \quad \leq \quad c\left(\sum_{i=1}^{N_{s>}} \frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P) + \frac{8}{P-1}\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}|+P) + \frac{8T_1}{P-1}\right).$$

Thus,

$$\begin{aligned}&\mathcal{E}(T^s_{s>})\\ \leq \quad & c\left(\mathcal{E}(\sum_{i=1}^{N_{s>}} \frac{E_{j_i}}{P}\log(|\mathcal{B}_{j_i}|+P)) + \frac{8}{P-1}\mathcal{E}(\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}|+P)) + \frac{8T_1}{P-1}\right).\end{aligned}$$

By Lemma 5.21, it follows that $\mathcal{E}(\sum_{i=1}^{N_{s>}} E_{j_i} \, \log(|\mathcal{B}_{j_i}|+P)) \in O(\sigma\log(PT_\infty))$, and $\mathcal{E}(\sum_{i=1}^{N_{s>}} \Sigma_{j_i} \, \log(|\mathcal{B}_{j_i}|+P)) \in O(\sigma\log(PT_\infty))$. Thus,

$$\mathcal{E}(T^s_{s>}) \quad \in \quad O(T_1/P + \sigma\log(PT_\infty)/P)\,.$$

We also need to bound the total number of timesteps in unsuccessful epochs. Let N_u be the random variable denoting the number of unsuccessful epochs. We prove the following lemma.

Lemma 5.26 *It holds that:*

$$\mathcal{E}(N_u) \quad \in \quad O\left(\frac{T_1}{P(e^{(P-1)/8}-2)\log P} + \frac{\sigma\log(PT_\infty)}{P(e^{(P-1)/8}-2)\log P}\right).$$

Proof: Let N_t be the random variable denoting the total number of stealing-concentrated epochs with at least P ready lists. Let $j_1,\ldots,j_{N_t}$ be these epochs. For each $i \in [N_t]$, we define the random variables

$$X_i \quad = \quad \begin{cases} 1 & \text{if epoch } j_i \text{ is successful} \\ 0 & \text{otherwise} \end{cases}$$

Recall that $\mathbf{Pr}[X_i = 1] \geq (1-2e^{-(P-1)/8})$. Moreover, $N_{s>} = \sum_{i=1}^{N_t} X_i$. We have:

$$\begin{aligned}\mathcal{E}(N_{s>} \mid N_t = n) \quad &= \quad \mathcal{E}(\sum_{i=1}^{n} X_i)\\ &\leq \quad n(1-2e^{-(P-1)/8})\,.\end{aligned}$$

Thus,

$$\begin{aligned}\mathcal{E}(N_{s>}) \quad &= \quad \mathcal{E}(\mathcal{E}(N_{s>} \mid N_t))\\ &= \quad \sum_n \mathcal{E}(N_{s>} \mid N_t = n)\mathbf{Pr}[N_t = n]\\ &\geq \quad (1-2e^{-(P-1)/8})\sum_n n\mathbf{Pr}[N_t = n]\\ &= \quad (1-2e^{-(P-1)/8})\mathcal{E}(N_t)\,,\end{aligned}$$

so that

$$\mathcal{E}(N_t) \;\leq\; \frac{\mathcal{E}(N_{s_>})}{1 - 2e^{-(P-1)/8}}$$

We define the 0/1-random variables $W_i = 1 - X_i$, so that $W_i = 1$, if the epoch is unsuccessful, while $W_i = 0$, otherwise. Clearly, $\mathbf{Pr}[W_i = 1] \leq 2e^{-(P-1)/8}$, while $N_u = \sum_{i=1}^{N_t} W_i$. We have:

$$\begin{aligned}
\mathcal{E}(N_u \mid N_t = n) &= \mathcal{E}(\sum_{i=1}^{n} W_i) \\
&\leq 2ne^{-(P-1)/8}\,.
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathcal{E}(N_u) &= \mathcal{E}(\mathcal{E}(N_u \mid N_t)) \\
&\leq \sum_n 2ne^{-(P-1)/8}\mathbf{Pr}[N_t = n] \\
&= 2e^{-(P-1)/8}\sum_n n\mathbf{Pr}[N_t = n] \\
&= 2e^{-(P-1)/8}\mathcal{E}(N_t)\,.
\end{aligned}$$

Since $\mathcal{E}(N_t) \leq \mathcal{E}(N_{s_>})/(1 - 2e^{-(P-1)/8})$, it holds that

$$\begin{aligned}
\mathcal{E}(N_u) &\leq \frac{2\mathcal{E}(N_{s_>})}{e^{(P-1)/8}(1 - 2e^{-(P-1)/8})} \\
&= \frac{2\mathcal{E}(N_{s_>})}{e^{(P-1)/8} - 2}\,.
\end{aligned}$$

By Lemma 5.25, $\mathcal{E}(N_{s_>}) \in O(T_1/(P\log P) + \sigma\log(PT_\infty)/(P\log P))$. Therefore, $\mathcal{E}(N_u) \in O(T_1/(P\log P(e^{(P-1)/8} - 2)) + \sigma\log(PT_\infty)/(P\log P(e^{(P-1)/8} - 2)))$, as needed. ■

Assume that $T^u_{s_>}$ is the random variable denoting the total number of timesteps spent for *unsuccessful* stealing-concentrated epochs with more than P ready lists. Let $j_1, \ldots, j_{N_u}$ be these epochs. Proposition 5.19 implies that each unsuccessful epoch j_i takes $O((E_{j_i}/P + 1)\log(|\mathcal{B}_{j_i}| + P)$ timesteps, so that there exists a constant $c' > 0$ such that

$$\begin{aligned}
T^u_{s_>} &\leq c'\sum_{i=1}^{N_u}(E_{j_i}/P + 1)\log(|\mathcal{B}_{j_i}| + P) \\
&= c'\sum_{i=1}^{N_u} E_{j_i}\log(|\mathcal{B}_{j_i}| + P)/P + c'\sum_{i=1}^{N_u}\log(|\mathcal{B}_{j_i}| + P)\,.
\end{aligned}$$

Thus,

$$\begin{aligned}
&\mathcal{E}(T^u_{s_>}) \\
&= c'\mathcal{E}(\sum_{i=1}^{N_u} E_{j_i}\log(|\mathcal{B}_{j_i}| + P))/P + c'\mathcal{E}(\sum_{i=1}^{N_u}\log(|\mathcal{B}_{j_i}| + P))\,.
\end{aligned}$$

By Lemma 5.21, it holds that $\mathcal{E}(\sum_{i=1}^{N_u} E_{j_i}\log(|\mathcal{B}_{j_i}|+P)) \in O(\sigma\log(PT_\infty))$, and $\mathcal{E}(\sum_{i=1}^{N_u}\log(|\mathcal{B}_{j_i}|+P)) \in O(\mathcal{E}(N_u)\log(PT_\infty))$, while by Lemma 5.26, $\mathcal{E}(N_u) \in O(T_1/(P\log P(e^{(P-1)/8}-2)) + \sigma\log(PT_\infty)/(P\log P(e^{(P-1)/8}-2)))$. Therefore, there exists some constant $c>0$ such that

$$\begin{aligned}
&\mathcal{E}(T^u_{s_>}) \\
\leq\ & c\left(\frac{\sigma\log(PT_\infty)}{P} + (\frac{T_1}{P\log P} + \frac{\sigma\log(PT_\infty)}{P\log P})\cdot\frac{1}{e^{(P-1)/8}-2}\log(PT_\infty)\right) \\
=\ & c\left(\frac{\sigma\log(PT_\infty)}{P} + (\frac{T_1}{P} + \frac{\sigma\log(PT_\infty)}{P})\cdot\frac{\log(PT_\infty)}{(e^{(P-1)/8}-2)\log P}\right).
\end{aligned}$$

Clearly, $\log(PT_\infty)/((e^{(P-1)/8}-2)\log P) \in O(1)$, whenever $P \geq \log_P \log T_\infty$. Thus, $\mathcal{E}(T^u_{s_>}) \in O(T_1/P + \sigma\log(PT_\infty)/P)$, whenever $P \geq \log_P \log T_\infty$.
Clearly, $T_{s_>} = T^s_{s_>} + T^u_{s_>}$, so that $\mathcal{E}(T_{s_>}) = \mathcal{E}(T^s_{s_>}) + \mathcal{E}(T^u_{s_>})$. Thus, we conclude that $\mathcal{E}(T_{s_>}) \in O(T_1/P + \sigma\log(PT_\infty)/P)$, whenever $P \geq \log_P \log T_\infty$.
At this point the proof of Lemma 5.27 is complete.

■

Clearly, $T_s = T_{s_<} + T_{s_>}$. Thus, by Lemmas 5.23 and 5.24, and by linearity of expectation, we have:

Lemma 5.27 $\mathcal{E}(T_s) \in O(T_1/P + \sigma\log(PT_\infty)/P)$.

Clearly, $T_{\mathsf{PSDF}}(G,P) = T_e + T_s$. Thus, by Lemmas 5.22 and 5.27, and by linearity of expectation, we conclude that:

$$\mathcal{E}(T_{\mathsf{PSDF}}(G,P)) \in O(T_1/P + \sigma\log(PT_\infty)/P + T_\infty\log(PT_\infty)).$$

At this point the proof of Theorem 5.20 is complete. ■

Contention

To prove our execution time bound we didn't account for contention caused by competing processors. We prove now that the processors do not cause any serious contention for accessing the working arrays.

Proposition 5.28 *Consider any multithreaded computation G, let $P>0$ be any integer and let α, $0<\alpha<1$, be any real. Assume that $\mathcal{X}_{\mathsf{PSDF}}(G,P)$ is any execution schedule produced by algorithm* PSDF *and let e be some arbitrary position of any of the arrays Σ, Π, Δ and E. Then, at most 3 processors may content for accessing e.*

Proof: We start by proving that if a processor p tries to access some position of the steal array Σ, it may content with at most two other processors. Let $\Sigma[i][j]$ be any position of Σ, where $1 \leq i \leq 2$ and $1 \leq j \leq (1-\alpha)P$. Clearly, whenever p accesses $\Sigma[i][j]$ it first has to read the toggle bit σ_j and change its state. Proposition 5.3 implies that at most 2 processors write column j of array Σ during some scheduling iteration. Thus, p may content with at most one more working processor on σ_j. However, p may also content with one scheduling processor that tries to update $\Sigma[i][j]$. Thus, a processor p contents with at most 2 more processors while accessing any element of the steal array.

We continue to prove that a processor p accessing some position of the deletion array Δ may content with at most 2 other processors. Proving the same claim for the stall array Π can be done in an analogous way.

Let $\Delta[i][j]$ be any position of Δ, where $1 \leq i \leq 3$ and $1 \leq j \leq (1-\alpha)P$. We proceed by case analysis. Assume first that $i = 3$. Recall that $\Delta[3][j]$ can be written only by the working processor j, once per scheduling iteration. However, processor j may content with a scheduling processor that tries to access $\Delta[3][j]$. Thus, in this case, j contents only with one scheduling processor. Assume now that $i < 3$. Proposition 5.3 implies that at most 2 working processors try to access column j of array Δ. Thus, a processor p accessing $\Delta[i][j]$ (or more specifically δ_j) may content with at most one other working processor. Additionally, p may content with one scheduling processor. Thus, p contents with at most 2 processors, as needed.

We finally prove that a processor p trying to update a list of the array E may content with at most 2 more processors. Let $E[i][j]$ be any position of E, where $1 \leq i \leq 3$ and $1 \leq j \leq (1-\alpha)P$. We proceed by case analysis. Assume first that $i = 3$. Recall that $E[3][j]$ is accessed only by the working processor j and the scheduling processors. Thus, processor j contents with at most one scheduling processor in accessing $E[3][j]$. Assume now that $i < 3$. We prove that a processor p trying to update list $E[i][j]$ contents only with at most 2 other processors. Clearly, p accesses list $E[i][j]$ only if $\lambda_p = i$ and $h_p = j$. Proposition 5.3 implies that at most 2 processors access column j of array E during a scheduling iteration. Thus, there might exist at most one more processor p' with $h_{p'} = j$. By the way λ_p, $\lambda_{p'}$ and the toggle bit ζ_j are manipulated, it must be the case that $\lambda_p \neq \lambda_{p'}$, so that processor p' access different elements of column j (that is, they access different lists). The two processors may content when they access ζ_j. Thus, p may content with at most one more working processor when it tries to access $E[i][j]$ (or more specifically ζ_j). However, p may also content with a scheduling processor that tries to access $E[i][j]$, thus resulting in a total number of 3 contending processors, as needed.

At this point the proof of the proposition is complete. ■

We would like to point out that under a non-blocking implementation of the ready lists, processors do not wait in queues when they try to steal from the same ready list. Thus, access to the active array does not cause any serious contention to the algorithm. We can thus conclude that accessing the working arrays do not cause any serious contention during the execution of PSDF[12]. We can conclude that the working processors do not cause any serious contention during the execution of the algorithm, or more generally accessing the working arrays do not introduce any serious contention to the algorithm.

[12]It is worth pointing out that in all previous works on depth-first scheduling algorithms [5, 6, 26], the working processors could lead to a serious amount of contention due to the fact that they all try to access the same end-point of some arrays.

6 Handling Large Allocations

We had 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. Since our algorithm is a depth-first algorithm, it handles large allocations by inserting a number of dummy threads. More specifically, it inserts before every allocation of $m > K$ bytes a binary fork tree of depth $\log(m/\log P)$ so that $m/\log P$ threads are created at its leaves. (Notice that for our algorithm it always holds that $K > \log P$, so that $m/K < m/\log P$). Each of the dummy threads performs a no-op that takes one timestep but it is treated as if it allocates K space (where K is the memory threshold at the timestep that the dummy thread is executed), so that a processor gives up its ready list and performs stealing after executing each of these dummy threads. After all these threads have been executed, a processor can proceed with the allocation of m bytes.

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). Since in our cost model an allocation of m bytes requires $O(\log m)$ depth, this transformation 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 the one presented in the next theorem.

Theorem 6.1 *For any computation G and any integer $P \geq \max\{3, \log\log T_\infty(G)\}$, it holds that:*

Space Bounds: (1) $\mathcal{E}(S_{\mathsf{PSDF}}(G,P)) \leq S_1 + O(pT_\infty \log(PT_\infty))$;

(2) *for any* $\varepsilon > 0$,

$$S_{\mathsf{PSDF}}(G,P) \quad \leq \quad S_1 + O(p(T_\infty + \ln(1/\varepsilon))\log(P(T_\infty + \ln(P(T_\infty + \ln(1/\varepsilon))/\varepsilon)))) ,$$

with probability at least $1 - \varepsilon$.

Time Bound: $\mathcal{E}(T_{\mathsf{PSDF}}(G,P)) \in O(T_1/P + (\sigma + S_K)\log(PT_\infty)/P + T_\infty \log(PT_\infty))$.

7 Conclusion

We have presented an online, space-efficient, low contention scheduling algorithm for the class of programs with write-once synchronization variables. The algorithm compines the depth-first approach with the work-scheduling paradigm and presents good locality and low contention.

We proved that the working processors do not cause any serious contention during the execution of the algorithm. The scheduling processors in our algorithm result also in less contention than in [6], since our algorithm achieves better scheduling granularity. Unfortunately, measuring the scheduling granularity of these algorithms in a quantitative way is not an easy task. None of the works presented thus far (including [25]) provides any such theoretical measure. However, it is not difficult to see that our algorithm achieves better scheduling granularity than the one in [6]. In our algorithm, a processor does not give up its ready list when threads spawn or every time a thread enables a stalled thread (as is the case for the algorithm presented in [6]), so that it is often the case that threads close together in the computation graph are scheduled at the same processor which results in better locality, while additionally the working tree is not accessed so often.

If the parallel computation is very fine-grained, the number of synchronizations σ can be as large as the work T_1, resulting in a running time of $O(\log(pT_\infty)(T_1/P+T_\infty))$. However, since synchronizations are expensive, there has been considerable work in reducing their number using compile-time analysis (see e.g., [20, 30]). Using these techniques the running time of our algorithm can probably be reduced. Moreover, the synchronization queues (which appear to be the only high-contention data structures used by our algorithm) can be implemented [6, Section 8] using asynchronous low-contention data structures such as counting networks [3] and diffracting trees [31].

Acknowledgments: We would like to thank Anil Kumar and Paul Spirakis for several useful discussions.

References

[1] N. S. Arora, R. D. Blumofe and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 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,* Vol. 11, No. 4, pp. 598–632, October 1989.

[3] J. Aspnes, M. Herlihy and N. Shavit, "Counting networks," *Journal of the ACM*, Vol. 41, No. 5, pp. 1020–1048, 1994.

[4] 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,* Vol. 21, No. 1, pp. 4–14, April 1994.

[5] G. E. Blelloch, P.B. Gibbons and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM,* Vol. 46, No. 2, pp. 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," *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 12–23, Newport, Rhode Island, June 1997.

[7] R. D. Blumofe, *Executing Multithreaded Programs Efficiently,* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 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," *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 297–308, Padua, Italy, June 1996.

[9] R. D. Blumofe and C. E. Leiserson, " Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM,* Vol. 46, No. 5, pp. 720–748, September 1999.

[10] R. Chandra, A. Gupta and J. L. Hennessy, "COOL: An object-based language for parallel programming," *IEEE Computer,* Vol. 27, No. 8, pp. 13–26, August 1994.

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

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

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

[14] P. Fatourou and P. Spirakis, "A New Scheduling Algorithm for General Strict Multithreaded Computations," *Proceedings of the 13rd International Symposium on DIStributed Computing,* pp. 297–311, Bratislava, Slovakia, September 1999.

[15] P. Fatourou and P. Spirakis, "Efficient Scheduling of Strict Multithreaded Computations," *Theory of Computing Systems Journal (TOCS),* to appear.

[16] J. Feo, D. Cann and R. Oldehoeft, "A Report on the Sisal Language Project," *Journal of Parallel and Distributed Computing,* Vol. 10, No. 4, pp. 349–366, December 1990.

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

[18] R. H. Halstead, "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 4, pp. 501–538, October 1985.

[19] High Performance Fortran Forum, *High Performance Fortran Language Specification version 1.0,* 1993.

[20] J. Hoch, D. Davenport, V. Grafe and K. Steele, "Compile-time partitioning of a non-strict language into sequential threads," *Proceedings of Symposium on Parallel and Distributed Computing,* December 1991.

[21] J. JáJá, *An Introduction to Parallel Algorithms,* Addison–Wesley Publishing Company Inc., USA, November 1992.

[22] D. Kranz, R. Halstead and E. Mohr, "Mul-T: A High-Performance Parallel Lisp," *Proceedings of the Conference on Programming Language Design and Implementation,* Portland, Oregon, June 1989.

[23] P. Mills, L. Nyland, J. Prins, J. Reif and R. Wagner, "Prototyping parallel and distributed programs in Proteus," UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

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

[25] G. Narlikar, "Scheduling Threads for Low Space Requirement and Good Locality," *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 83–95, Saint-Malo, France, June 1999.

[26] G. Narlikar, *Space-Efficient Scheduling for Parallel, Multithreaded Computations,* PhD Thesis, CMU-CS-99-119, School of Computer Science, Carniege Mellon University, Pittsburgh, May 1999.

[27] G. Narlikar and G. Blelloch, "Pthreads for Dynamic and Irregular Parallelism," *Proceedings of SuperComputing '98: High Performance Networking and Computing,* November 1998.

[28] G. Narlikar and G. Blelloch, "Space-Efficient Implementation of Nested Parallelism," *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* June 1997.

[29] M. Rinard, D. Scales and M. Lam, "Jade: A high-level, machine-independent language for parallel programming," *IEEE Computer*, Vol. 26, No. 6, pp. 28–38, June 1993.

[30] K. Schauser, D. Culler and S. Goldstein, "Separation constraint partitioning–a new algorithm for partitioning non-strict programs into sequential threads," *Proceeding of the Symposium on Principles of Programming Languages,* pp. 259–271, January 1995.

[31] N. Shavit, E. Upfal and A. Zemach, "A steady state analysis of diffracting trees," *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 33–41, Padua, June 1996.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Im Stadtwald
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?

MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable "open-world" desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid E-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph
MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates

MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number