A New Scheduling Algorithm for General Strict Multithreaded Computations*

Panagiota Fatourou[†] *Paul Spirakis*[‡]

Abstract

Three important performance parameters of scheduling algorithms for *multithreaded computations* on parallel computers are the required *space*, their *execution time*, and the *communication cost* incurred during the course of an execution; the first is characterized by the amount of storage needed for an execution, the second is the total number of steps needed for executing all threads, while the last is the amount of communication incurred for keeping more than one processors busy and resolving any kind of dependencies that may exist between threads executing on different processors. In this paper, we study the problem of efficiently scheduling a wide class of multithreaded computations, called *strict*; that is, computations in which all dependencies from a thread are directed to the thread's ancestors in the computation tree. We present the first scheduling algorithm which applies to *any* strict multithreaded computations and is provably good in terms of all three parameters. The algorithm is distributed, randomized and follows the *work-stealing* paradigm. Our analysis applies for both shared-memory and distributed-memory parallel machines.

More specifically, the expected execution time of our algorithm for scheduling any strict multithreaded computation on P processors is $O(T_1/P + hT_\infty)$, where T_1 is the minimum serial execution time, T_∞ is the minimum execution time with an infinite number of processors, and h is the maximum "distance" in the *computation tree* between any two threads that need to communicate. Furthermore, the total space required during the execution is $O(S_1 P)$, where S_1 is the space required by a serial computer to execute the computation, while the expected communication cost incurred by our algorithm is $O(PhT_\infty(1+n_d)S_{max})$, where n_d is the maximum number of dependencies entering any thread and S_{max} is the largest amount of storage needed for the execution of any specific thread of the computation.

*This work has been supported in part by the European Union's ESPRIT Long Term Research Project ALCOM-IT (contract # 20244).

[†]**Contact author.** Department of Computer Engineering and Informatics, University of Patras, Patras, Greece & Computer Technology Institute, Patras, Greece. Postal address: Computer Technology Institute, P. O. Box 1122, 26110 Patras, Greece, Fax. +30 61 993 973, E-mail: `faturu@cti.gr`

[‡]Department of Computer Engineering and Informatics, University of Patras, Patras, Greece & Computer Technology Institute, Patras, Greece. E-mail: `spirakis@cti.gr`

1 Introduction

Dynamically growing multithreaded computations are nowadays quite common for parallel computers. Multithreaded models of parallel computation have typically been proposed as a general approach to model dynamic, unstructured parallelism (see e.g., [17, 19, 26]) and have been employed by several parallel programming languages [12, 13, 14, 22]. To specify parallelism, a thread can spawn child threads. Additionally, a thread may synchronize with some or all of its (direct or indirect) descendants by suspending its execution until a descendant reaches a specific point of computation. A multithreaded computation is identified with a directed acyclic graph, henceforth abbreviated as dag. The nodes of this dag represent instructions of threads, while there are three kinds of different edges, *continue edges* for organizing instructions into threads, *spawn edges* for representing spawning of threads from other threads, and *dependency edges* declaring data or synchronization dependencies among different threads. Spawn edges organize threads into a rooted tree, called *spawn tree*.

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. Apparently, this is not a desirable task to be undertaken by a programmer. On the other hand transferring the control of these scheduling decisions to the run time system might not be wise, unless the system guarantees that it will make good scheduling decisions in order to execute the program efficiently. A good scheduling technique must ensure that enough threads remain active to keep the processors busy, while at the same time, the concurrently active threads must be within limits in order to control the (dynamic) memory needed. Moreover, in order to reduce the communication among processors, one should try to maintain related threads on the same processor. Apparently, designing a scheduler to achieve all of the above goals is not a trivial task.

Two scheduling paradigms have been considered in the past, *work-sharing* and *work-stealing*. In work-sharing, overutilized processors try to migrate some threads to other (hopefully underutilized) processors. On the contrary, in the work-stealing paradigm, underutilized processors "steal" work from other processors. The work-stealing paradigm dates back at least as far as Burton and Sleep's research [11] on parallel execution of functional programs and Halstead's implementation of Multilisp [18]. Since then a lot of work has been done in this direction (see e.g., [1, 4, 5, 6, 7, 8, 15]).

Three significant performance parameters of any scheduling algorithm for multithreaded computations are the required space, their execution time and the communication cost incurred by them. The *execution time* is the total time needed by the algorithm to execute the instructions of all threads composing the computation. Each thread requires a certain amount of memory for storing its data. The *space complexity* of a scheduling algorithm for the execution of a multithreaded computation is the maximum total storage needed by all processors during the course of the execution. The execution of a multithreaded computation on a distributed-memory parallel computer incurs also communication cost. This communication is required mainly for the "distribution" of threads among different processors (so that to keep more than one processor working), as well as for resolving dependencies among threads resided on different processors. The *communication complexity* of a scheduling algorithm for the execution of a multithreaded computation is the total amount of communication incurred during the course of the execution.

An algorithm achieves *linear speedup* if its execution time with P processors is P times faster than the optimal execution time on a one-processor computer. Since a processor can execute only one instruction at each time step, an algorithm that achieves linear speedup is optimal in terms of execution time. An algorithm uses *linear expansion of memory* if its space requirements are not more than P times the space requirements for the execution of the same computation on a one-processor computer. There exist very simple and common multithreaded computations that require linear expansion of memory in order to achieve linear speedup (see e.g., [5, Section 2.3]). Hence, a scheduling algorithm that uses linear expansion of memory is arguably efficient in terms of space complexity.

Say that a multithreaded computation is *fully strict* when all dependencies from each thread go to the thread's parent. Apparently, the class of fully strict multithreaded computations is a rather restricted one. On the opposite extreme, a *general* multithreaded computation allows arbitrary dependencies between threads. Clearly, any multithreaded computation belongs to the class of general computations. There is, in addition, an important middle ground between fully strict and general mulithreaded computations: those in which dependencies from any thread are directed to the thread's ancestors in the spawn tree. Call these computations *strict*. Apparently, the class of strict multithreaded computations is much wider than the class of fully strict ones. Clearly, the first class encompasses all distributed multithreaded applications in which communication between threads and *any* of their ancestors is required.

Indeed, there exists a technique, known as *strictification*, for reducing a wide class of multithreaded computations, which are called *depth-first**, into fully strict ones. This is done by adding suitable dummy nodes and edges into the instruction graph. However, this technique has several disadvantages. First, it can dramatically reduce the average parallelism by increasing the length of the maximum path in the instruction graph. Moreover, it may produce a large communication overhead in case of distributed memory machines. In the original instruction graph, communication between threads is expressed by dependency edges; threads that communicate a lot should be placed on the same processor. Apparently, by strictifying the computation, information about related threads is reduced, since several "unrelated" threads may need to participate for resolving a dependency. On the other hand, strictification requires the involvement of the programmer (for taking care of dummy nodes and dependency edges), which is not desirable in several cases. For all these reasons, strictification is not considered an efficient solution. It is thus interesting to discover other techniques or algorithms that schedules in a provably efficient way more general than fully strict multithreaded computations.

Blumofe and Leiserson [7] have proved that there exists no scheduling algorithm for general multithreaded computations to achieve both linear speedup and linear expansion of memory. They have presented a multithreaded computation for which every algorithm can not achieve not even a factor of two speedup without avoiding space requirements to grow as a function of the serial execution time. However, they have proved that by restricting to the class of strict multithreaded computations, there exist efficient execution schedules that both achieve linear

*A *depth-first* computation is one in which the "left-to-right" search of instructions in the computation always visits all of any given instruction's predecessors in the instruction graph before it visits the given instruction.

speedup and use linear expansion of memory. Thus, these results imply that if one wants to find a scheduling algorithm with good execution time and memory complexity, restricting to strict multithreaded computations does the job.

In a pioneering work, Blumofe and Leiserson [6] have considered the class of fully strict multithreaded computations and they have presented the first provably good work-stealing scheduler (in terms of all three performance parameters) for such computations. However, the class of fully strict multithreaded computations is very restricted, compared to the more general class of strict computations. Thus, an interesting question left open by their work is the existence of a provably good scheduling algorithm for (general) strict multithreaded computations. Blumofe and Leiserson [6, Section 7] have pointed out that generalizing their analysis to work for general strict computations is indeed both important and difficult.

In this paper, we show that it is, in fact, possible to enjoy good performance properties for this much wider class of multithreaded computations. Our result significantly enhances the class of multithreaded computations that can be scheduled efficiently in terms of both execution time and memory complexity. More specifically, we present a distributed, work-stealing scheduling algorithm for (general) strict multithreaded computations and we prove good bounds on its performance. Our algorithm enjoyes nice locality properties, while simultaneously it achieves to schedule together related threads. Moreover, we prove that our algorithm is efficient in terms of execution time, as well as space complexity. Our analysis applies not only for shared memory computers, but also for distributed memory systems. More significantly, we provide bounds on the communication complexity of our algorithm for such systems.

Our algorithm achieves expected execution time $O(T_1/P + hT_\infty)$, where T_1 is the optimal execution time with one processor, T_∞ is the length of a path of maximum length in the instruction dag, and h is the *dependency height* of the computation; that is, h is the maximum "distance" in the spawn tree between any two threads that need to communicate during the course of the execution. Clearly, no algorithm can achieve execution time better than T_1/P since each processor can execute only one task at a step. Moreover, no algorithm can achieve execution time less than T_∞, since none of the instructions on this path can be executed in parallel with any other instruction on the path. Notice that our algorithm achieves linear speedup, whenever $P \leq T_1/(hT_\infty)$, so that $hT_\infty \leq T_1/P$.

The space complexity of our algorithm is $O(S_1 P)$, where S_1 is the optimal space complexity with one processor. Thus, our algorithm uses *linear expansion of memory*. Recall that an algorithm that uses linear expansion of memory is arguably efficient in terms of space complexity.

The expected communication complexity of our algorithm is $O(PhT_\infty(1+n_d)S_{max})$, where $S_{\max}$ is the maximum size of storage needed by any thread of the computation and n_d is the maximum number of dependency edges entering any thread. Wu and Kung have proved in [27] that there exist fully strict multithreaded computations, such that any algorithm that achieves linear speedup incurs total communication at least $O(PT_\infty S_{max})$. This lower bound implies that scheduling algorithms for fully strict multithreaded computations that achieve this amount of communication are optimal in terms of communication cost. Our communication bound diverges by a factor of h from this lower bound. However, we argue that this divergence is not unexpected, since h appears to be a measure of the degree of "strictness" of multithreaded computations. Since for fully strict multithreaded computations $h = 1$, this divergence of our communication bound from the lower bound provided for fully strict multithreaded computations appears to us natural.

We also prove that for any $\epsilon > 0$, with probablity at least $1-\epsilon$ the algorithm's execution time is $O(T_1/P + hT_\infty + \log P + \log(1/\epsilon))$, while its communication complexity is $O(P(hT_\infty + \log(1/\epsilon))(1+n_d)S_{max})$ with probability again at least $1-\epsilon$.

Substantial research (see e.g., [1, 20, 24, 28]) has been reported in the literature concerning the scheduling of multithreaded computations, ignoring though space requirements and communication costs. Burton shows in [10] how to limit space in certain parallel computations without causing deadlock. More recently, Burton [9] has developed and analyzed a scheduling algorithm with provably good time and space bounds. Blelloch *et al.* [3, 4] have also recently developed and analyzed scheduling algorithms with provably good time and space bounds for languages with nested fine-grained parallelism (that is, languages that lead to series-parallel DAGs). All these algorithms are analyzed only for shared-memory machines and do not account for communication cost. On the opposite, our analysis holds even for distributed-memory machines; more significantly, we provide bounds on the communication complexity of our algorithm for such machines.

Fatourou and Spirakis considered in [15] the case of k-strict conputations, that is, computations on which any dependency from a thread goes to some of the k ancestors of the thread in the activation tree. They present two scheduling algorithms for k-strict multithreaded computations. Their algorithms work in a different way than the algorithm presented in this paper and require knowledge of k. On the contrary, the algorithm presented in this paper is based on new ideas, which employ timestamps and maintains extra and more complicated data structures in order to keep track of related threads, that is threads that communicate a lot and thus should be placed on the same processor. Proving particular properties on the structure of these data structures appears to be a major challenge of our analysis. These properties does not hold for the algorithms presented in [15], so that those algorithms are not appropriate for the more general case of strict computations.

Recently, Arora *et al.* [1] have proved that the work-stealing algorithm presented in [6] can be analyzed assuming general multithreaded computations. However, their analysis provides only an execution time bound and no bounds for the space complexity. In contrast, our algorithm provides bounds on both execution time and space complexity, albeit for the less general class of strict multithreaded computations. Additionally, the analysis provided in [1] applies only for shared memory computers, while our analysis applies even for distributed memory machines; more significantly, we provide bounds on the communication complexity of our algorithm for such machines.

The rest of this paper is organized as follows. Section 2 includes definitions and some preliminary facts, while Section 3 presents our algorithm, exhibits fundamental properties maintained by the algorithm, and provides space, execution and communication bounds on its performance.

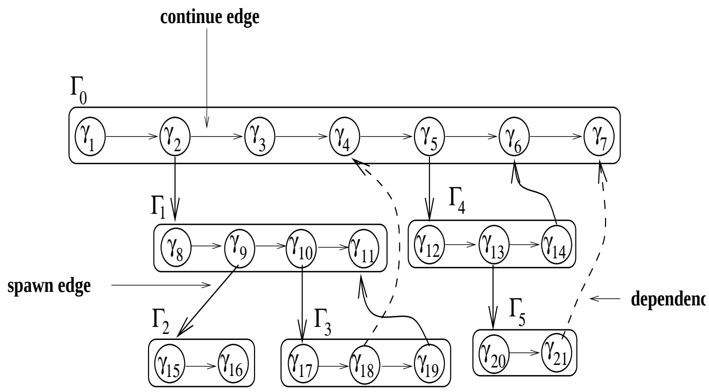

Figure 1: A multithreaded computation.

2 The Model

In this section, we describe our model. Our definitions closely follows the ones presented in [6, 15].

A *multithreaded computation* is modeled as a graph G, which is called the *instruction graph* of the computation. A multithreaded computation is composed of a set of *threads*, each of which contains unit-time *tasks*. Tasks are executed in some sequential order. The nodes of the instruction graph represent tasks of the multithreaded computation. Tasks are connected to each other via *continue edges* that determine the order in which the tasks of a thread are executed. During the course of a thread's execution, the thread may create, or *spawn*, *child* threads. A thread can spawn as many children as it likes to. If a task γ of a thread Γ_1 spawns another thread Γ_2, a *spawn edge* begins from task γ and ends at the first task of thread Γ_2 in the instruction graph. Spawn edges organize the threads into a rooted tree, which is called the *spawn tree* of the computation. Call the thread that corresponds to the root of this tree the *root thread* of the multithreaded computation. Denote by $\delta(G)$ the maximum length of any path between any two nodes in the spawn tree of G. A thread with no children is called a *leaf thread*. In addition to the continue and spawn edges, the instruction graph may also contain *dependency edges*. Dependency edges model data dependencies, like e.g., producer-consumer dependencies, and allow threads to synchronize. If the execution of a thread arrives at a "consuming" task, before the execution of the corresponding "producing" task, the execution of the consuming thread *stalls*. Once the producing task executes, the dependency is *resolved* and the consuming task is *enabled* to proceed with its execution. For any multithreaded computation G, we denote by $E_d(G)$ the set of the dependency edges of G.

Figure 1 presents a multithreaded computation. A thread in this graph is represented by a block of circles, where each circle represents a particular task of the thread. The horizontal edges in Figure 1 are continue edges. Spawn edges are represented by downward-pointing edges. Thus, thread Γ_0 is the parent of threads Γ_1 and Γ_4. Dependency edges are represented by curved arrows between tasks of different threads. For example, task γ_4 of thread Γ_0 is a consuming task. It needs a data item that is produced by task γ_{18} of thread Γ_3. Thus, task γ_4 can not be executed until after task γ_{18}.

Clearly, the execution of a multithreaded computation is possible only if dependency edges do not produce cycles in the instruction graph. Thus, this graph must be a directed acyclic graph or *dag*[†]. We denote by n_d the maximum number of dependency edges entering any specific thread. A thread *dies* when all its tasks have been executed; a *dead* thread is one that has died. A spawned thread that is not *dead* is *alive*.

A multithreaded computation is said to be *strict* if all dependencies from one thread goes to ancestors of the thread in the spawn tree. A *fully-strict multithreaded computation* is one in which all dependencies from a thread go to the thread's parent. Multithreaded computations that contain any other kind of dependencies are called *non-strict* or *general*. Our example computation of Figure 1 is a strict computation. If we remove the dashed edges from the multithreaded computation of Figure 1, it becomes a fully strict multithreaded computation.

For any thread Γ of a multithreaded computation, we define its *height* to be the distance of thread Γ from the root thread in the spawn tree. Clearly, the height of the root thread is 0. We can partition the spawn tree into levels, where level 0 contains only the root thread, level 1 contains all threads of height 1, and generally level i contains all threads of height i. For any two integers $i, j \geq 0$, we say that level i is *higher* than level j if $i < j$. For any thread Γ, we denote by $level(\Gamma)$ the level that Γ is located in the spawn tree of G. Consider any thread Γ located at level j in the spawn tree. For any integer $i \leq j$, we define the *i-ancestor* of Γ to be the ancestor of the thread located at level $j - i$ in the spawn tree. For any integer $i \geq 0$, the *i-descendant* of Γ is defined in the natural way.

For any multithreaded computation G and any two threads $\Gamma, \Gamma' \in G$ connected by a dependency edge $e \in E_d(G)$, we define the *dependency height* of edge e, denoted by h_e, to be $h_e = |level(\Gamma) - level(\Gamma')|$. For any multithreaded computation G, the dependency height of G, denoted $h(G)$, is defined to be $h(G) = \max_{e \in E_d(G)}\{h_e\}$. Clearly, the dependency height of any multithreaded computation G is at most $\delta(G)$. In the rest of this paper, we use the notation h instead of $h(G)$, that is, we ommit G whenever it is clear from the context.

An execution schedule for a multithreaded computation G, denoted $\mathcal{X}(G, P)$, determines which processors of the parallel computer execute which instructions at each step. A *valid* execution schedule must satisfy all constraints imposed by the continue, spawn and dependency edges of the instruction dag of a multithreaded computation. At any given step t of an execution schedule, a task is *ready* if all of its predecessors in the instruction dag have been executed. Only ready tasks may be executed. We say that an execution schedule maintains the *busy-leaves property* if at each time step, every leaf thread has a processor working on it.

When a task spawns a thread, it allocates an *activation frame*, that is a block of memory, for use by the newly spawned thread. We denote by $S_{max}(G)$ the maximum size of the activation frame of any thread of a multithreaded computation G. We denote by $S_1(G)$ the minimum amount of space required for the execution of G on a one processor machine.

The *work* of a computation G is the total number of tasks in the computation. The *dag depth of a task* in G is the length of the longest path of the instruction graph that terminates

[†]We often refer to the instruction graph of a computation as *instruction dag*.

at the task. The *dag depth of* G is the maximum dag depth of any of its tasks. We denote by $T_1(G)$ the work of the computation, since a one-processor computer can only execute one task at any time step. 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 any multithreaded computation G, a P-processor ($P > 1$) execution schedule for G may incur an amount of communication when running in a distributed way. Apart from the amount of communication needed for the "distribution" of threads among the different processors, communication may also be required for resolving dependencies among threads residing on different processors. We denote by $C(\mathcal{X}(G, P))$ the total communication required for the execution of the multithreaded computation G by the execution schedule $\mathcal{X}(G, P)$.

A *scheduling algorithm* for a multithreaded computation decides which processors execute which tasks at every time step; that is, for any multithreaded computation G and any integer $P > 0$, a scheduling algorithm Alg takes G and P as input and outputs a schedule $\mathcal{X}_{\mathsf{Alg}}(G, P)$.

Since more than one threads may be simultaneously ready in a single processor, a mechanism for scheduling threads within a single processor should be provided by any algorithm. Call this mechanism the *internal scheduler*. In order to keep more than one processors working, a scheduling algorithm must dynamically distribute work among processors. Thus, a basic component of any scheduling algorithm is the *external scheduler*, whose job is to schedule threads across different processors. A *work-stealing* external scheduler works as follows. When a processor runs out of work, it becomes a *thief* and steals work from a *victim* processor. The decision of who is going to be the victim processor may be taken either deterministically or in a randomized way, yielding to either deterministic or randomized algorithms, respectively. An algorithm should also provide a mechanism for deciding what actions are made when a dependency is resolved. Call this mechanism the *dependency resolver*.

In this paper, we consider only distributed scheduling algorithms. In a *distributed* scheduling algorithm a copy of the above three mechanisms runs on each processor independently.

If any of the three components of a scheduling algorithm works in a randomized way, the algorithm produces a probability distribution over all execution schedules $\mathcal{X}(G, P)$. In this work, we concentrate on scheduling algorithms with randomized, work-stealing external schedulers. For each such algorithm Alg, we denote by $T_{\mathsf{Alg}}(G, P)$, $S_{\mathsf{Alg}}(G, P)$ and $C_{\mathsf{Alg}}(G, P)$ the random variables denoting the execution time, the total space required, that is the *space complexity*, and the communication incurred, that is the *communication complexity*, during the P-processor execution of a multithreaded computation G by Alg.

The *work of* Alg on a multithreaded computation G, denoted $I_{\mathsf{Alg}}(G, P)$, is defined to be the number of tasks (instructions) that Alg executes during the execution of G on a P-processor parallel computer. Notice that the work of any scheduling algorithm, whether randomized or not, is always equal to $T_1(G)$. We define the *stealing time of* Alg, denoted $L_{\mathsf{Alg}}(G, P)$, to be the random variable expressing the number of steal attempts occured during the P-processor execution of Alg on the multithreaded computation G. The *waiting time of* Alg, denoted $W_{\mathsf{Alg}}(G, P)$, is the random variable expressing the total time that processors have to wait due to contention caused by stealing at the ready lists of processors.

[‡]It is natural that in most computations G, $T_\infty(G) >> \delta(G)$.

We continue to describe fundamental analytical tools, which can be proved useful for analysing the complexity of work-stealing, scheduling algorithms.

For any multithreaded computation G, the *augmented instruction dag* G' contains the original graph as a subgraph, but it has some extra edges, which are called *critical edges*. For every set of tasks γ_i, γ_j and γ_m, such that $\langle \gamma_i, \gamma_j \rangle$ is a spawn edge and $\langle \gamma_i, \gamma_m \rangle$ is a continue edge, the critical edge $\langle \gamma_m, \gamma_j \rangle$ is also an edge of G'.

For any time step t during the execution of a multithreaded computation G on a P-processor parallel computer according to some execution schedule $\mathcal{X}_{\mathsf{Alg}}(G, P)$, an unexecuted task γ is said to be *critical* at time step t, if all tasks γ', such that there exists a path from γ' to γ in G', have been executed by time step t. A *critical path* of tasks in G', denoted $U(G) = \langle \gamma_1, \ldots, \gamma_L \rangle$, is defined to be a path in G' which is maximal. We say that a critical path $U(G) = \langle \gamma_1, \ldots, \gamma_L \rangle$ *occurs* during the execution of a multithreaded computation G according to some valid execution schedule $\mathcal{X}(G, P)$, if one and only one of its tasks is critical at each time step of the execution.

A *round of steal attempts* during the execution of some multithreaded computation G is a set of at least $6hP$ but fewer than $6hP + P$ consecutive steal attempts such that if a steal attempt initiated at time step t occurs in a particular round, then all other steal attempts initiated at time step t are also in the same round. All steal attempts that occur during an execution can be partitioned into rounds as follows. The first round contains all steal attempts initiated at time steps $1, 2, \ldots, t_1$, where t_1 is the earliest time step such that at least $6hP$ steal attempts were initiated at or before t_1. In general, if the ith round ends at time step t_i, then the $(i+1)$st round begins at time step $t_i + 1$ and ends at the earliest time step $t_{i+1} > t_i + 1$ such that at least $6hP$ steal attempts were initiated at time steps between $t_i + 1$ and t_{i+1}, inclusive. We say that a given round of steal attempts *occurs* while some instruction γ is critical if all of the steal attempts that comprise the round are initiated at time steps when γ is critical; that is, γ must be critical throughout the entire round.

For any multithreaded computation G, a *delay sequence*, denoted by $\mathcal{D}(G)$, is a 3-tuple $\langle U(G), R, \Pi \rangle$ satisfying the following conditions: (1) $U(G) = \langle \gamma_1, \gamma_2, \ldots, \gamma_L \rangle$ is a critical path; (2) R is a positive integer; (3) $\Pi = \langle \pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L \rangle$ is a partition of the integer R (that is, $R = \sum_{i=1}^{L} (\pi_i + \pi'_i)$), such that $\pi'_i \in \{0, 1\}$, for each $i \in [L]$. We define the ith *group* of rounds to be the π_i consecutive rounds starting after the r_ith round, where $r_i = \sum_{j=1}^{i-1} (\pi_j + \pi'_j)$.

Assume that Alg is any work-stealing, scheduling algorithm and let $\mathcal{X}_{\mathsf{Alg}}(G, P)$ be any execution schedule produced by Alg for the execution of G on a P-processor parallel computer. A delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi) \rangle$ such that $U(G) = \langle \gamma_1, \ldots, \gamma_L \rangle$ and $\Pi = \langle \pi_1, \pi'_1, \ldots, \pi_L, \pi'_L \rangle$, is said to *occur* during the execution of G by some execution schedule $\mathcal{X}_{\mathsf{Alg}}(G, P)$, if the critical path $U(G)$ occurs during the execution and for each $i \in [L]$, all π_i steal attempt rounds in the ith group occur while task γ_i is critical; that is, γ_i must be critical throughout all π_i rounds.

3 A Randomized Distributed Scheduling Algorithm

3.1 Description

The algorithm is online, distributed and works in a randomized, work-stealing fashion. Each processor maintains a data structure of ready threads with two endpoints, `top` and `bottom`, which is called *ready list*. The ready list is always sorted, according to the height of the threads it contains[§]. Threads of higher height occupy positions in the ready list closer to the bottom end. We further associate to each thread a timestamp. When a thread is spawned it receives a timestamp equal to the value of the local clock of the processor in whose ready list is placed.

Some of the elements of the ready list of each processor are linked to one another in a way that they comprise a doubly linked list, called the *enabling list*. We will explain which elements of the ready list participate in the construction of the enabling list below. The enabling list has two endpoints, `start` and `end`, and it is always sorted according to the timestamps of threads it contains. Threads of larger timestamps reside in positions closer to the end of the enabling list. Each processor p maintains a pointer, called $\mathtt{uptr}_p$, which points to an appropriate element of p's read list. We will explain how each processor maintains pointer `uptr` below.

Algorithm SRWS is a distributed algorithm; thus, each one of the processors runs its own internal scheduler, external scheduler and dependency resolver. However, these components operate in the same way on every processor. The algorithm starts with all ready lists empty, except from the ready list of the processor where the root thread is initiated. All enabling lists are initially empty, and all `uptr` pointers are initially `null`.

The internal scheduler of some processor p pushes and pops threads in its processor's ready list from the bottom, like if it were a stack. More specifically, the internal scheduler of p removes the bottomost thread of its processor's ready list and starts work on it. If this bottomost thread belongs to the enabling list, the internal scheduler updates the appropriate pointers of the enabling list. If pointer $\mathtt{uptr}_p$ points to this thread, $\mathtt{uptr}_p$ is updated to `null`. The internal scheduler works on this bottomost thread until it spawns, stalls, dies or enables a stalled thread, in which case it performs according to the following rules: (1) If a thread Γ_i spawns a child thread Γ_j, then Γ_i is placed on the bottom of the ready list, and the internal scheduler continues work on Γ_j. If pointer $\mathtt{uptr}_p$ was `null` before the spawn, it is updated to point to Γ_i. (2) If a thread Γ_i enables a stalled thread Γ_j, the dependency resolver runs. (3) If a thread Γ_i stalls or dies, its internal scheduler checks its ready list. If the list contains any threads, then the internal scheduler removes and begins work on the bottomost thread. If the bottomost thread belongs to the enabling list, the internal scheduler updates the appropriate pointers of the enabling list. If pointer $\mathtt{uptr}_p$ points to the bottomost thread, $\mathtt{uptr}_p$ is updated to `null`. If the ready list is empty, the external scheduler runs, and tries to obtain work from other processors. In case a thread enables another thread and dies, the dependency resolver runs first, before the actions for dying.

The dependency resolver of each processor works as follows. If a thread Γ_i enables a stalled thread Γ_j, the now ready thread Γ_j is placed in the proper position, according to its priority, at the ready list of Γ_i's processor. The timestamp of thread Γ_j takes the value of the local clock at the current time step of the processor, in whose ready list Γ_i resides. Moreover, Γ_j participates in the construction of the enabling list (that is, it becomes one of its elements)[¶].

The external scheduler of any processor works as a thief and tries to steal work from a victim

[§]Notice that the parent of the thread can easily notify its children about their height. Thus, the notification of each thread about its height can be performed in a straightforward way.

[¶]Notice that the timestamp of Γ_j is the maximum timestamp among all threads residing in the enabling list. Thus, we simply link this thread at the end of the enabling list.

processor, which has been chosen uniformly at random. The external scheduler queries the ready list of the victim processor[||] and if it is non-empty it steals some thread of the processor's ready list. To choose which thread to steal the thief processor works as follows. The timestamp of the first thread in the enabling list is compared to the timestamp of the thread pointed by pointer `uptr` and the thread of smallest timestamp between the two is stolen. If pointer `start` is `null`, the thread pointed by `uptr` is stolen. If pointer `uptr` is `null`, the thread pointed by `start` is stolen. If the thread pointed by `uptr` is stolen, pointer `uptr` is updated to point to the next element towards the bottomost thread of p's ready list. If there is no such element, it takes the value `null`. If the ready list of the victim processor is empty, the thief processor selects another victim processor uniformly at random and this procedure is repeatedly applied till a victim processor with a non-empty ready list is found. When the external scheduler of a processor achieves to obtain work, the control pass to the internal scheduler of the processor again.

3.2 Properties

For each integer $n \geq 1$, denote $[n] = \{1, \ldots, n\}$. Our first proposition presents important properties of pointer `uptr`. Roughly speaking, Proposition C.2 asserts that the threads comprising the enabling list of some processor p occupy the topmost positions in its ready list; moreover, pointer $\texttt{uptr}_p$ points the topmost element in p's ready list that does not belong in the enabling list.

Proposition 3.1 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Let Γ_0 be the thread that p is working on, let n be the number of threads in p's ready list and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the threads in p's ready list ordered from bottom to top, so that Γ_1 is bottomost and Γ_n is topmost. If pointer* `uptr` *is not* `null` *and points to thread Γ_m, threads in p's ready list satisfy the following properties:*

(1) *for all $i = 1, \ldots, m$, (a) thread Γ_{i-1} is a child of thread Γ_i in the spawn tree; (b) thread Γ_{i-1} has larger timestamp than thread Γ_i; (c) thread Γ_i has not been worked on since it spawned thread Γ_{i-1}, so that thread Γ_i is not an element of the enabling list;*

(2) *If $n > m$, for all $i = m + 1, \ldots, n - 1, n$, thread Γ_i is an element of the enabling list.*

Next proposition establishes that a thread can be stolen from some processor's ready list only after all its ancestor threads of smaller timestamp in p's ready list have been stolen.

Proposition 3.2 *Let G be any multithreaded computation and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step t during*

[||]In message-passing systems, a stealing attempt to the ready list of a victim processor can be implemented by sending a message to this processor. The external scheduler of each processor is then responsible to process this kind of messages.

the course of an execution of G by some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$. Let Γ, Γ' be threads in p's ready list such that thread Γ is an ancestor of thread Γ' and Γ has a smaller timestamp than Γ'. Then, thread Γ' can not be stolen from p's ready list, unless thread Γ is stolen.

The following proposition presents fundamental properties of the structure of the ready list of any processor. These properties are very important for proving the performance bounds of our algorithm.

Proposition 3.3 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$. Let Γ_0 be the thread that p is working on, let n be the number of threads in p's ready list and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the threads in p's ready list ordered from bottom to top, so that Γ_1 is bottomost and Γ_n is topmost. If $n > h+1$, then the threads in p's ready list satisfy the following properties: (1) for all $i = 1, \ldots, n-h$, thread Γ_{i-1} is a child of thread Γ_i in the spawn tree; (2) for all $i = 1, \ldots, n-h-1$, thread Γ_i has not been worked on since it spawned Γ_{i-1}; (3) if $i \in \{h+1, \ldots, n\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l has larger timestamp than thread Γ_i.*

Sketch of proof: The proof is by induction on execution time. For the basis case, where $t = t_0$, all claims hold vacously, since the only ready thread is the root thread. For proving the induction step we study several cases on the action taken by the task executed at some processor p one time step before the considered time step. The most difficult case to consider is when the task enables a stalled thread. We use Propositions 3.1 and 3.2 to prove a collection of statements, the most important of which are the following: (1) the enabled thread is placed in one of the h topmost positions of p's ready list; (2) for any thread Γ, which has been placed in p's ready list at some time step t for the last time, no l-descendant of Γ, where $l > h$, exists in p's ready list at time step t; moreover, (3) processor p does not commence work-stealing after time step t, while simultaneously (4) it is not possible any such descendant to be placed in p's ready list by enabling. Thus, all these descendants should be spawned in p's read list. We then prove that due to particular properties of both processor's external stealers and dependency resolvers, as well as due to the way timestamps and pointers `uptr` are updated, no such thread can be stolen from p's ready list. We use the above facts to prove the stated claims. ■

We continue to prove important properties of critical tasks. We start by proving that during the execution of a multithreaded computation by algorithm SRWS, a critical task, residing in the ready list of some processor p, is either an element of p's enabling list or pointer $\mathtt{uptr}_p$ points to it.

Proposition 3.4 *Consider a strict multithreaded computation G and let $P > 1$ be any integer. At every time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$, each critical task must be the ready task of a thread that either is an element of the enabling list of some processor p or pointer $\mathtt{uptr}_p$ points to it.*

The following proposition states that during the execution of any strict multithreaded computation by Algorithm SRWS on a parallel computer with P processors, any critical task executes after at most $(2h+1)$ steal requests have been serviced on the ready list of the processor that the thread containing this task resides.

Proposition 3.5 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that γ is the critical task of a thread Γ residing in the ready list of processor $p \in [P]$, at some time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Then, after at most $(2h+1)$ steal requests have been serviced on processor p, task γ executes.*

Sketch of proof: We prove that there exist at most h ancestors and at most h descendants with larger timestamp than the thread that γ belongs, in one processor's ready list. Moreover, we prove that after $2h$ steal attempts have been serviced by processor p, all these threads have been stolen and either γ has already been executed or it is to be stolen next, as needed. ■

We next prove that during the execution of any strict multithreaded computation by Algorithm SRWS on a P-processor parallel computer, the probability that $r > 1$ rounds of steal attempts occur while any particular task is critical is exponentially decreasing with hr.

Proposition 3.6 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then, for any task γ and any integer $r > 1$, the probability that r rounds occur during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$, while γ is critical is at most e^{-2hr}.*

We finally bound the probability that a particular delay sequence occurs during the execution of a strict multithreaded computation by Algorithm SRWS.

Proposition 3.7 *Consider any multithreaded computation G and let $P > 1$ be any integer. Assume that $\mathcal{D}(G) = \langle U(G), R, \Pi \rangle$ is any arbitrary delay sequence such that $U(G)$ is a critical path of length L. Then, the following holds for the probability that $\mathcal{D}(G)$ occurs during the execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$: $\mathbf{P}[\mathcal{D}(G) occurs] \leq e^{-2h(R-2L)}$.*

3.3 Complexity

In this section, we present upper bounds on the performance of Algorithm SRWS.

Theorem 3.8 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then, (1) $S_{\mathsf{SRWS}}(G, P) \leq S_1 P$; (2) for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the following holds: $T_{\mathsf{SRWS}}(G, P) \in O(T_1/P + hT_\infty + \log P + \log(1/\epsilon))$; moreover, $\mathcal{E}(T_{\mathsf{SRWS}}(G, P)) \in O(T_1/P + hT_\infty)$; (3) for any $\epsilon > 0$, with probability at least $1 - \epsilon$, $C_{\mathsf{SRWS}}(G, P) \in O(P(hT_\infty + \log(1/\epsilon))(1 + n_d)S_{max})$; moreover, $\mathcal{E}(C_{\mathsf{SRWS}}(G, P)) \in O(PhT_\infty(1 + n_d)S_{max})$.*

Sketch of proof: We prove that all execution schedules produced by SRWS maintain the busy-leaves property. It has been proved [5] that all execution schedules that maintain the busy-leaves property use only linear expansion of memory. In order to bound the execution time of our algorithm, we bound seperately its work, its stealing time and its waiting time. Then, we add up these three factors and divide by P. Clearly, the work of SRWS on G is $T_1(G)$. For bounding the stealing time, we prove that during the execution of any multithreaded computation if a large number of steal attempts are initiated, a delay sequence occurs. However, Proposition 3.7 implies that the probability of a delay sequency to occur is exponentially decreasing with the number of rounds of steal attempts. Thus, with high probability, a large number of steal attempts does not occur. We use the bound derived for *SRWS*'s steal time to bound its communication complexity. In order to bound the waiting time of *SRWS*, we use a combinatorial balls and bins game introduced and anlyzed in [6]. ∎

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,* Puerto Vallarta, Mexico, June–July 1998.

[2] G. E. Blelloch, "Programming parallel algorithms," *Proceedings of Dartmouth Institute for Advanced Graduate Studies (DAGS) Symposium on Parallel Computation,* pp. 11–18, Hanover, New Hampshire, June 1992.

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

[4] 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,* Newport, Rhode Island, pp. 12–23, June 1997.

[5] R. D. Blumofe, "Executing Multithreaded Programs Efficiently," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[6] R. D. Blumofe and C. E. Leiserson, " Scheduling Multithreaded Computations by Work Stealing," *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science,* pp. 356–368, 1994 (full version to appear in JACM).

[7] R. D. Blumofe and C. E. Leiserson, "Space-Efficient Scheduling of Multithreaded Computation," *SIAM Journal on Computing,* Vol. 27, No. 1, pp. 202–229, February 1998.

[8] R. D. Blumofe and D. S. Park, "Scheduling large-scale parallel computations on networks of workstations," *Proceedings of the 3rd International Symposium on High Performance Distributed Computing,* pp. 96–105, San Francisco, California, August 1994.

[9] F. W. Burton, "Guaranteeing good memory bounds for parallel programs," *IEEE Transactions on Software Engineering,* Vol. 22, No. 10, October 1996.

[10] F. W. Burton, "Storage management in virtual tree machines," *IEEE Transactions on Computers,* Vol. 37, No. 3, pp. 321–328, March 1988.

[11] F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," *Proceedings of the Conference on Functional Programming Languages and Computer Architecture,* pp. 187–194, Portsmouth, New Hampshire, October 1981.

[12] M. C. Carlisle, A. Rogers, J. H. Reppy and L. J. Handren, "Early experiences with Olden," *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing,* Portland, Oregon, August 1993.

[13] 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.

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

[15] P. Fatourou and P. Spirakis, "Scheduling Algorithms for Strict Multithreaded Computations," *Proceedings of the 10th Internbational Sumposium on Algorithms and Computation,* pp. 407–416, Osaka, Japan, 1996.

[16] W. Feller, *An Introduction to Probability Theory and its Application,* Second Edition, Vol. 1, John Wiley & Sons, Inc., 1957.

[17] V. G. Grafe and J. E. Hoch, "The Epsilon-2 hybrid dataflow architecture," *Proceedings of the 35th IEEE International Computer Conference (COMPCON '90),* pp. 88–93, San Francisco, California, February 1990.

[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] R. A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," *Proceedings of the 15th Annual International Symposium on Computer Architecture,* pp. 131–140, Honolulu, Hawaii, May 1988.

[20] R. M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch-and-bound computation," *Journal of the ACM,* Vol. 40, No. 3, pp. 765–789, July 1993.

[21] P. Liu, W. Aielo and S. Bhatt, "An atomic model for message passing," *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93),* pp. 154–163, Velen, Germany, June 1993.

[22] R. S. Nikhil, "Cid: A parallel, shared-memory C for distributed-memory machines," *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing,* August 1994.

[23] A. Ranade, "How to emulate shared memory," *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science,* pp. 185–194, Los Angeles, California, October 1987.

[24] L. Rudolph, M. Slivkin-Allalouf and E. Upfal, "A simple load balancing scheme for task allocation in parallel machines," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 237–245, Hilton Head, California, July 1991.

[25] V. A. Saletore, J. Jacob and M. Padala, "Parallel computations on the CHARM heterogenous workstation cluster," *Proceedings of the 3rd International Symposium on High Performance Distributed Computing,* pp. 203–210, San Francisco, California, August 1994.

[26] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi and Y. Koumura, "Thread-based programming for the EM-4 hybrid dataflow machine," *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pp. 146–155, Gold Coast, Australia, May 1992.

[27] I. C. Wu and H. T. Kung, "Communication complexity for parallel divide-and-conquer," *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS '91),* pp. 151–162, San Juan, Puerto Rico, October 1991.

[28] Y. Zhang, *Parallel Algorithms for Combinatorial Search Problems,* PhD Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Technical Report UCB/CSD 89/543, November 1989.

[29] Y. Zhang and A. Ortynski, "The efficiency of randomized parallel backtrack search," *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing,* Dallas, Texas, October 1994.

Appendix

A Mathematical Foundation

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

Stirling's formula states that $n! \approx \sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n}$. We continue with a simple combinatorial fact:

Lemma A.1 *For any integers $n, r > 0$, the following holds:*

$$\binom{n+r}{r} \leq e^{n+r} .$$

Proof: Clearly,

$$\begin{aligned}
\binom{n+r}{r} &= \frac{(n+r)!}{n!r!} \\
&= \frac{(r+1)\ldots(r+n)}{n!} \\
&\leq \frac{(n+r)^n}{n!} \\
&\approx \frac{(n+r)^n}{\sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n}} \\
& \quad \text{(by Stirling formula)} \\
&\leq e^n \left(\frac{n+r}{n}\right)^n \\
&= e^n \left(1 + \frac{r}{n}\right)^n \\
&\leq e^{n+r} ,
\end{aligned}$$

as needed. ■

Repeated independent trials are called *Bernoulli trials* in case there are only two possible outcomes for each trial and their probabilities remain the same throughout the trials. Usually, these two probabilities are denoted by p and q. The outcome with probability p is refered as *success*, while the other as *failure*. Clearly, p and q are non-negative and $p + q = 1$.

Let $b(r; n, p)$ be the probability that n Bernoulli trials with probabilities p for success and $q = 1 - p$ for failure result in r successes and $n - r$ failures. Then, clearly,

$$b(r; n, p) = \binom{n}{r} p^r q^{n-r} .$$

Denote $\mathbf{S}_n$ the random variable representing the number of successes in n trials; clearly, $b(r;n,p) = \mathbf{P}[\mathbf{S}_n = r]$. The probability distribution $b(r;n,p)$ of $\mathbf{S}_n$ is called the *binomial distribution.*

For the binomial distribution, the following is true (see e.g., Feller [16, Chapter 6, Section 3]):

Lemma A.2 *For any integer $k < (n+1)p$,*

$$\sum_{r=1}^{k} b(r;n,p) \quad \le \quad b(k;n,p)\,\frac{(n-k+1)p}{(n+1)p-k}\,.$$

A.1 The (P,M)-Recycling Game

Our presentation closely follows the one in [6], where the (P,M)-recycling game is introduced and analyzed.

The (P,M)-recycling game is a combinatorial game, where balls are tossed into bins by an adversary. The number of balls in the game equals P; it is also equal to the number of bins. The parameter M is the total number of ball tosses executed by the adversary. Initially, all balls are in a reservoir seperate from the P bins and the number of ball tosses is 0. The adversary works in steps, executing the following operations per step.

(1) The adversary removes some arbitrarily chosen balls from the reservoir (possibly all and possibly none). Then, for each of these balls, the adversary selects one of the P bins uniformly and independently at random, and tosses the ball into it. The total number of ball tosses should not exceed M. If the adversary has already performed M ball tosses this operation is skipped.

(2) The adversary inspects each of the P bins in turn, and for each bin that contains at least one ball, the adversary removes one arbitrary ball of the bin and returns it to the reservoir.

The game ends when M ball tosses have been made and all balls have beem removed and placed back in the reservoir. After each step t of the game, there are some number b_t of balls left in the bins. Let the *total delay* of the game, denoted by D, be the random variable denoting the sum of b_t for all time steps t of the game; that is,

$$D \quad = \quad \sum_{i=1}^{T} b_t\,,$$

where T is the total number of steps in the game. The goal of the adversary is to make the total delay as large as possible. It can be proved, however, that despite the adversary's choices about which balls to toss into bins and which to return in the reservoir, the total delay is unlikely to be large.

Proposition A.3 (Blumofe and Leiserson [6]) *Let D be the total delay of any arbitrary instance of the (P,M)-recycling game. Then,*

(1) *for any* $\epsilon > 0$*, with probability at least* $1-\epsilon$*, it holds that* $D \in O(M + P\log P + P\log(1/\epsilon))$*;*

(2) $\mathcal{E}(D) \in O(M)$*.*

Thus, Proposition A.1 states that, for any $\epsilon > 0$, with probability at least $1-\epsilon$ the total delay in the (P, M)-recycling game is $O(M + P\log P + P\log(1/\epsilon))$, while the expected total delay is $O(M)$.

B The Model

B.1 Execution Schedules

Next proposition states that for any strict multithreaded computation G, the space needed for an execution of G according to some valid execution schedule that maintains the busy-leaves property is bounded by $PS_1(G)$; that is, execution schedules that maintain the busy leaves property use *linear expansion of memory.*

Proposition B.1 (Blumofe [5]) *Consider a strict multithreaded computation* G*, and let* $\mathcal{X}(G, P)$ *be any* P*-processor valid execution schedule of* G*, for any* $P > 0$*. Then, if* $\mathcal{X}(G, P)$ *maintains the busy-leaves property,*

$$S(\mathcal{X}(G, P)) \quad \leq \quad PS_1(G)\,.$$

For any multithreaded computation G, if we add up the work, the stealing time and the waiting time of any algorithm Alg for the execution of G on a P-processor parallel computer and divide by P, we will obtain its execution time.

Lemma B.2 (Blumofe and Leiserson [6]) *Consider any multithreaded computation* G *and let* $P > 1$ *be any integer. Assume that* Alg *is any work-stealing scheduling algorithm for* G*. Then,*

$$T_{\mathsf{Alg}}(G, P) \quad = \quad \frac{I_{\mathsf{Alg}}(G, P) + L_{\mathsf{Alg}}(G, P) + W_{\mathsf{Alg}}(G, P)}{P}\,.$$

B.2 Analytical Tools

B.2.1 Augmented Instruction Graph

Figure 2 presents the augmented instruction dag G' that corresponds to our example multithreaded computation of Figure 1. In Figure 2, critical edges are shown dashed.

For every set of tasks γ_i, γ_j and γ_m, such that $\langle \gamma_i, \gamma_j \rangle$ is a spawn edge and $\langle \gamma_i, \gamma_m \rangle$ is a continue edge, the critical edge $\langle \gamma_m, \gamma_j \rangle$ is also an edge of G'. We make the technical assumption that instruction γ_m has no incoming dependency edges, so that G' is a dag. If a cycle is created, a new "phantom" task can be inserted between γ_i and γ_m. Notice that the insertion of these "dummy" nodes does not affect the asymptotic bounds presented in this paper.

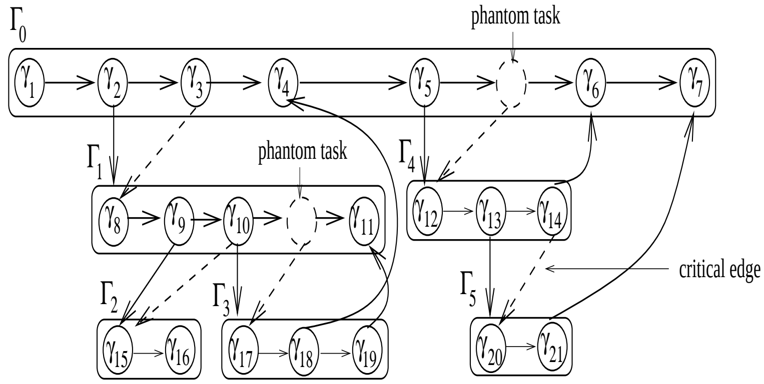

Figure 2: The augmented instruction dag of our example multithreaded computation of Figure 1.

Next observation is due to Blumofe and Leiserson [6]; it is an immediate consequence of the way graph G' is constructed.

Observation B.1 *For any multithreaded computation G, $T_\infty(G') \leq 2T_\infty(G)$.*

Consider three threads Γ_i, Γ, Γ_m of G such that threads Γ_i, Γ are ancestors of thread Γ_m and Γ is a child of Γ_i. Assume further that γ_m is any task of thread Γ_m, γ is the first task of thread Γ and γ_i is the next task to the task of thread Γ_i that spawns thread Γ. Clearly, there exists a path from task γ to task γ_m in the instruction dag. Since the augmented graph contains also the edge $\langle \gamma_i, \gamma \rangle$, there exists a path from task γ_i to task γ_m in G', as stated by the following observation.

Observation B.2 *Consider any multithreaded computation G and let γ be a task of a thread Γ_i such that γ spawns a new thread. Assume that γ_m is any task of a thread Γ_m such that Γ_m is a descendant of thread Γ_i in the spawn tree. Then, if γ_i is the next task to γ in thread Γ_i, there exists a path connecting γ_i and γ_m in G'.*

In our example augmented instruction dag of Figure 2 threads Γ_i, Γ_m could be, for instance, threads Γ_0, Γ_2, respectively, while tasks $\gamma, \gamma_i, \gamma_m$ could be tasks $\gamma_2, \gamma_3, \gamma_{16}$, respectively. Clearly, there exists a path from task γ_3 to γ_{16} in the augmented instruction dag, as stated by Observation B.2.

B.2.2 Critical Tasks - Critical Paths

We first prove a proposition stating that during the course of an execution of any multithreaded computation a critical path occurs.

Proposition B.3 *Consider any multithreaded computation G. Then, in an execution of G according to some arbitrary valid execution schedule $\mathcal{X}(G,P)$, a critical path $U(G)$ occurs.*

Proof: For constructing the critical path $U(G)$ we use corresponding arguments as those presented in [5, Section 4, Lemma 4.6]. We work backwards from the instruction executed last, which we denote by γ'_1. Let γ'_{l_1} denote the (not necessarily immediate) predecessor task of γ'_1 in G' with the latest execution time. Let $\langle\gamma'_{l_1},\ldots,\gamma'_1\rangle$ denote a directed path from γ'_{l_1} to γ'_1. We iterate this construction, inductively, back to the first task of the root thread. At the ith iteration we have an instruction γ'_{l_i} and a directed path in G' from γ'_{l_i} to γ'_1. We let $\gamma'_{l_{i+1}}$ denote the predecessor of γ'_{l_i} in G' with the latest execution time, and let $\langle\gamma'_{l_{i+1}},\ldots,\gamma'_{l_i}\rangle$ denote a directed path from $\gamma'_{l_{i+1}}$ to γ'_{l_i} in G'. We finish our construction when we get to an iteration m in which γ'_{l_m} is the first instruction of the root thread. Our desired critical path is then $U(G)=\langle\gamma_1,\gamma_2,\ldots,\gamma_L\rangle$, where $L=l_m$ and $\gamma_i=\gamma'_{L-i+1}$, for $i\in[L]$. Moreover, for each $i\in[m]$, let $\gamma_{l_i}=\gamma'_{l_{m-i+1}}$. Denote by $\gamma_{l_{m+1}}=\gamma_L$; that is, $\gamma_{l_{m+1}}$ is the last task of the execution. Notice that γ_{l_1} is the first task of the root thread; thus, γ_{l_1} is the first task of the execution.

We now prove that at every time step of the execution, one and only one of the γ_{l_i} is critical; therefore, the described critical path occurs. For simplicity of proof, we denote by t_T the time step of the execution that γ_L executes; that is, t_T is the last time step of the execution. Moreover, for each $i\in[m]$, we denote by t_{l_i} the time step that task γ_{l_i} executes. Notice that t_{l_1} is the first time step of the execution. We further denote by t_{l_0} time step t_0; that is, t_{l_0} is a time step before the beginning of the execution. We prove, by induction on i, that for each $i\in[m+1]$, task γ_{l_i} becomes a critical task at time step $t_{l_{i-1}}$ (when task $\gamma_{l_{i-1}}$ executes) and remains critical for all time steps $t_{l_{i-1}+1},\ldots,t_{l_i-1}$, while no other task in $U(G)$ is critical in the same time interval.

For the basis case that $i=1$ the claim holds vacously. Since γ_{l_1} is the first task of the execution, it does not have incoming edges and thus it is a critical task till the time step that it is executed, that is time step t_{l_1}.

Assume now that for some integer i such that $2\leq i<m+1$ the following hold:

(1) task $\gamma_{l_{i-1}}$ becomes critical at time step $t_{l_{i-2}}$ and remains critical for all time steps $t_{l_{i-2}+1},\ldots,t_{l_{i-1}-1}$;

(2) no other task in $U(G)$ is critical in the same time interval.

For the induction step we prove that

(1) γ_{l_i} becomes critical at time step $t_{l_{i-1}}$ and remains critical for all time steps $t_{l_{i-1}+1},\ldots,t_{l_i-1}$;

(2) no other task in $U(G)$ is critical in the same time interval.

At time step $t_{l_{i-1}}$ task $\gamma_{l_{i-1}}$ is executed. By construction of $U(G)$, task $\gamma_{l_{i-1}}$ is the predecessor of γ_{l_i} with the latest execution time. Thus, all other predecessors of γ_{l_i} must have been executed by time step $t_{l_{i-1}}$. Hence, after the execution of $\gamma_{l_{i-1}}$, all predecessors of γ_{l_i} (including $\gamma_{l_{i-1}}$) have been executed, which means that γ_{l_i} becomes critical at time step $t_{l_{i-1}}$. Clearly, all tasks preceeding γ_{l_i} in path $U(G)$ have already been executed by time step $t_{l_{i-1}}$, so that none of them can be critical after time step $t_{l_{i-1}}$. Moreover, all tasks in path U(G) successing task γ_{l_i} are descendants of γ_{l_i}. Thus, none of them can be critical (because they all have a common predecessor, namely task γ_{l_i}, that has not yet been executed). Thus, γ_{l_i} remains critical till time step t_{l_i-1}, while no other task in $U(G)$ is critical in the time interval $t_{l_{i-1}}, \ldots, t_{l_i-1}$, which completes the proof of the induction step.

Since t_{l_1} is the first time step of the execution and $t_{l_{m+1}} = t_T$ is the last time step of the execution, critical path $U(G)$ occurs during the execution. ■

B.2.3 Rounds of Steal Attempts - Delay Sequences

The following observation bounds the total number of steal attempt rounds that belong to any group; it is immediate, since Π is a partition of R and for each $i \in [L]$, $\pi'_i \in \{0, 1\}$**.

Observation B.3 *For any delay sequnce $\mathcal{D}(G) = \langle U(G), R, \Pi\rangle$ such that $U(G) = \langle \gamma_1, \ldots, \gamma_L\rangle$ and $\Pi = \langle \pi_1, \pi'_1, \ldots, \pi_L, \pi'_L\rangle$,*

$$\sum_{i=1}^{L} \pi_i \quad \geq \quad R - L\,.$$

We denote by $\#_{\mathcal{D}(G)}$ the number of possible delay sequences in G'. The following technical lemma, proved by Blumofe and Leiserson [6], provides an upper bound on the number of delay sequences for any multithreaded computation G.

Lemma B.4 (Blumofe and Leiserson [6]) *For any multithreaded computation G,*

$$\#_{\mathcal{D}(G)} \quad \leq \quad (d(G)+1)^{2T_\infty(G)} \begin{pmatrix} 4T_\infty(G) + R \\ R \end{pmatrix}.$$

Blumofe and Leiserson [6] have proved that if during the course of an execution of a fully strict multithreaded computation, both a critical path $U(G)$ occur and at least R steal attempt rounds for fully strict multithreaded computations take place, then some delay sequence $D(G) = \langle U(G), R, \Pi\rangle$ occurs. The same arguments can be applied for proving that this is also true for any multithreaded computation, as stated by the following proposition.

Proposition B.5 *Consider any multithreaded computation G and let $P > 0$ be any integer. Assume that during the execution of G by any arbitrary valid execution schedule $\mathcal{X}(G, P)$ a critical path $U(G)$ occurs, and also R steal attempt rounds take place. Then, some delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi\rangle$ occurs.*

**A similar observation for fully strict multithreaded computations and rounds of steal attempts for such computations is presented in [6].

Roughly speaking, for proving the above proposition it suffices to construct the partition $\Pi = \langle \pi_1, \pi'_1, \ldots, \pi_L, \pi'_L \rangle$, so that the π_i rounds, $i \in [L]$, that comprise the ith group all occur while the instruction γ_i of the occuring critical path $U(G) = \langle \gamma_1, \ldots, \gamma_L \rangle$ is critical. This can be done by letting each π_i be the number of rounds that occur while γ_i is critical and by letting each π'_i be the number of rounds that begin while γ_i is critical but do not end until after γ_i is executed[††]. Since at most one round can begin while γ_i is critical and end after γ_i is executed, it is $\pi'_i \in \{0, 1\}$. Moreover, the definition of the occurence of a critical path implies that the π_i rounds that comprise the ith group all occur while only instruction γ_i is critical and thus Π is indeed a partition.

We finally prove that if during the execution of a multithreaded computation G a large number of steal attempts occur then a delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi \rangle$ occurs.

Proposition B.6 *Consider any multithreaded computation G and let $P > 1$ be any integer. Then, if at least $(6hP + P)R$ steal attempts occur during the execution of G by any arbitrary valid execution schedule, a delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi \rangle$ occurs.*

Proof: Proposition B.3 implies that there exists a critical path $U(G)$ that occurs during the execution of G by any arbitrary valid execution schedule $\mathcal{X}(G, P)$. Moreover, with at least $(6hP + P)R$ steal attempts, at least R rounds of steal attempts take place during the execution. Thus, Proposition B.5 immediately implies that a delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi \rangle$ occurs, as needed. ∎

C A Randomized Distributed Scheduling Algorithm

C.1 Properties

We assume that concurrent accesses to the ready list of one processor are serially queued by the external scheduler of the processor (as in the atomic message-passing model of [21]). We also assume that if concurrent steal requests are made to a ready list in one time step, one request is satisfied but the others are queued by the external scheduler for processing on next time steps.

For any multithreaded computation G, we denote by $d(G)$ the degree of G.

Any execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$ produced by Algorithm SRWS respects the continue edges of the instruction graph, since it executes each thread from its first to its last task. Moreover, $\mathcal{X}_{\mathsf{SRWS}}(G, P)$ respects the spawn edges, since threads that have not yet been spawned are not ready, and thus they do not exist in any ready list. It can be easily verified that $\mathcal{X}_{\mathsf{SRWS}}(G, P)$ also respects the dependency edges of the instruction dag. This is so, because whenever the execution reaches a consuming task requesting a data item that is not yet available, the execution of the thread containing the consuming task stalls. The stalled thread becomes ready again only after the producing task executes. Thus, we conclude that all execution schedules produced by Algorithm SRWS are valid execution schedules.

[††]For each $i \in [L]$, each π'_i indicates whether γ_i is critical at a beginning of a round but gets executed before the round ends. Such a round can not be part of any group, because no instruction is critical throughout.

C.1.1 Structural Properties of the Ready List

Our first lemma is an immediate consequence of the priority nature of the ready list.

Lemma C.1 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$. Let Γ_0 be the thread that p is working on, n be the number of threads in p's ready list and $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the threads in p's ready list ordered from bottom to top, so that Γ_1 is bottomost and Γ_n is topmost. Then, for all $i = 1, \ldots, n$, thread Γ_{i-1} is a descendant of Γ_i in the spawn tree.*

The following proposition presents important properties of pointer `uptr`. Roughly speaking, Proposition C.2 asserts that the ready list of any processor can be partitioned in two parts. The bottomost part contains consecutive pairs of threads that satisfy the parent-child relationship, while the elements of the enabling list resides in the top positions of a processor's ready list. The bound between the two lists is determined by the thread pointed by `uptr`.

Proposition C.2 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$. Let Γ_0 be the thread that p is working on, let n be the number of threads in p's ready list and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the threads in p's ready list ordered from bottom to top, so that Γ_1 is bottomost and Γ_n is topmost. If pointer `uptr` is not `null` and points to thread Γ_m, threads in p's ready list satisfy the following properties:*

(1) *for all $i = 1, \ldots, m$,*

   (a) *thread Γ_{i-1} is a child of thread Γ_i in the spawn tree;*

   (b) *thread Γ_{i-1} has larger timestamp than thread Γ_i;*

   (c) *thread Γ_i has not been worked on since it spawned thread Γ_{i-1}, so that thread Γ_i is not an element of the enabling list;*

(2) *If $n > m$, for all $i = m+1, \ldots, n-1, n$, thread Γ_i is an element of the enabling list.*

Proof: By induction on the execution time. Let $t_0, t_1, \ldots, t_T$ be the time steps of the execution and let $p \in [P]$ be any arbitrary processor. For simplicity of proof we use the following notation. For each $j \in [T]$, let Γ_0^j be the thread that processor p works on at time step t_j. Notice that Γ_0^j might be `null`, if the processor does not have work to do at time step t_j (e.g., if it tries to steal some thread at t_j). Moreover, assume that the ready list of processor p contains $n_j \geq 0$ ready threads at time step t_j, which we denote by $\Gamma_1^j, \ldots, \Gamma_{n_j}^j$, ordered from bottom to top, so that Γ_1^j is bottomost and $\Gamma_{n_j}^j$ is topmost. Furthrmore, for any integer $j \in [T]$, and any arbitrary thread Γ of p's ready list at time step t_j, we denote by $timestamp_j(\Gamma)$, the timestamp of thread Γ at time step t_j; moreover, we denote by $\Gamma_{m_j}^j$ the thread that pointer $\mathtt{uptr}_p$ points at time step t_j. Clearly, if $\mathtt{uptr}_p \neq \mathtt{null}$ at time step t_j, $m_j > 0$; otherwise, we assume that $m_j = 0$.

Basis Case

For the basis case where $t = t_0$ the claims hold vacously, since the only ready thread is the root thread, which resides in the ready list of some processor, while all other ready lists are empty.

Induction Hypothesis

Assume now that the claims hold for some time step t_{j-1} such that $j \in [T]$; that is, if $m_{j-1} > 0$, the threads in p's ready list satisfy the following properties:

(1) for all $i = 1, \ldots, m_{j-1}$,

(a) thread Γ_{i-1}^{j-1} is a child of thread Γ_i^{j-1} in the spawn tree;

(b) $\mathit{timestamp}_{j-1}(\Gamma_{i-1}^{j-1}) > \mathit{timestamp}_{j-1}(\Gamma_i^{j-1})$;

(c) thread Γ_i^{j-1} has not been worked on since it spawned thread Γ_{i-1}^{j-1}, so that thread Γ_i^{j-1} is not an element of the enabling list.

(2) If $n_{j-1} > m_{j-1}$, for all $i = m_{j-1} + 1, \ldots, n_{j-1} - 1, n_{j-1}$, thread Γ_i^{j-1} is an element of the enabling list.

Induction Step

We will prove that the claims hold for t_j; that is, if $m_j > 0$, the threads in p's ready list satisfy the following properties at time step t_j:

(1) for each $i = 1, \ldots, m_j$,

(a) thread Γ_{i-1}^{j} is a child of thread Γ_i^{j} in the spawn tree;

(b) $\mathit{timestamp}_{j}(\Gamma_{i-1}^{j}) > \mathit{timestamp}_{j}(\Gamma_i^{j})$;

(c) thread Γ_i^{j} has not been worked on since it spawned thread Γ_{i-1}^{j}, so that thread Γ_i^{j} is not an element of the enabling list.

(2) If $n_j > m_j$, for each $i = n_j, n_j - 1, ..., m_j + 1$, thread Γ_i^j is an element of the enabling list.

The following cases should be considered for the action taken by the task, let it be γ, that executes at time step t_{j-1}: (1) γ is a "simple" instruction, (2) γ spawns a new thread, (3) thread Γ_0^j stalls or dies, and (4) γ enables a stalled thread. Moreover, the case that another processor steals a thread from p's ready list should be considered. Thus, we proceed by case analysis.

1. We consider first the case where γ executes a "simple" instruction.

   Then, $n_j = n_{j-1}$, $m_j = m_{j-1}$, $\Gamma_0^j = \Gamma_0^{j-1}$ and for each $i \in [n_j]$ we have that $\Gamma_i^j = \Gamma_i^{j-1}$, and $\mathit{timestamp}_j(\Gamma_i^j) = \mathit{timestamp}_{j-1}(\Gamma_i^{j-1})$, so that induction hypothesis immediately implies that all claims hold.

2. We now consider the case where the executed task spawns a thread.

   Clearly, $n_j = n_{j-1} + 1$. According to the way the internal scheduler works, the newly created thread becomes the active thread that the processor works on, while the thread that performed the spawning is placed back in the bottomost position of the ready list. Thus, for each $i \in [n_j]$ we have that $\Gamma_i^j = \Gamma_{i-1}^{j-1}$, while $m_j = m_{j-1} + 1$. Thread Γ_0^j is spawned by $\Gamma_1^j = \Gamma_0^{j-1}$ at time step t_{j-1}, so that thread Γ_0^j is a child of thread Γ_1^j. Clearly, Γ_0^j has larger timestamp than any other thread in p's ready list at t_j, while thread Γ_1^j has not been worked on since it spawned Γ_0^j because the spawn only just occured. Moreover, no changes on the enabling list occur at time step t_{j-1}, so that induction hypothesis implies that all claims hold.

3. We continue to consider the case where thread Γ_0^{j-1} stalls or dies.

   If $n_{j-1} = 0$, the ready list is empty and processor p commences work-stealing. Independently of whether it will acquire work or not all claims hold vacously. Otherwise, $n_{j-1} > 0$, so that p's ready list is not empty. The internal scheduler pops the bottomost thread from the ready list and the processor starts work on it.

   We proceed to prove (1). Clearly, if $m_{j-1} \leq 1$ then $m_j = 0$, so that (1) holds vacously. Assume now that $m_j > 1$. Then, we have $n_j = n_{j-1} - 1$, $m_j = m_{j-1}$, and $\Gamma_0^j = \Gamma_1^{j-1}$. Moreover, for all $i = 1, \ldots n_j = n_{j-1} - 1$, we have $\Gamma_i^j = \Gamma_{i+1}^{j-1}$. Thus, by induction hypothesis (1), all claims hold.

   We continue to prove (2). Assume first that $m_{j-1} \neq 1$, so that $m_j = m_{j-1}$. Induction hypothesis implies that for all $i = m_{j-1} + 1, \ldots, n_{j-1}$, thread Γ_i^{j-1} is an element of the enabling list. Recall that $n_j = n_{j-1} - 1$, and for each $i = 1, \ldots, n_j$, $\Gamma_i^j = \Gamma_{i+1}^{j-1}$. Thus, for all $i = m_j + 1, \ldots, n_j$, thread Γ_i^j is an element of the enabling list, as needed.

   Assume now that $m_{j-1} = 1$. Then, $m_j = 0$. Induction hypothesis implies that for all $i = 2, \ldots, n_{j-1}$, thread Γ_i^{j-1} is an element of the enabling list. Recall that $n_j = n_{j-1} - 1$, and for each $i = 1, \ldots, n_j$, $\Gamma_i^j = \Gamma_{i+1}^{j-1}$. Thus, for all $i = 1, \ldots, n_j$, thread Γ_i^j is an element of the enabling list. Since $m_j = 0$, this completes the proof of (2).

4. We now consider the case that γ enables a stalled thread.

   Let Γ_e^j be the enabled thread. Clearly, $n_j = n_{j-1} + 1$.

   We start by proving (1). Assume first that $m_{j-1} = 0$. Then, $m_j = 0$, so that (1) holds vacously.

   Assume now that $m_{j-1} > 0$. By induction hypothesis (1/a), for each $i \in 1, \ldots m_{j-1}$, thread Γ_{i-1} is a child of thread Γ_i. Thus, the enabled thread is some ancestor of Γ_m^{j-1}, so that $m_j = m_{j-1}$ and induction hypothesis implies that (1) holds.

   We continue to prove (2). Recall that the enabled thread is an ancestor of Γ_m^{j-1} and $m_j = m_{j-1}$. By induction hypothesis (2) for each $i \in n_{j-1}, \ldots m_{j-1} + 1$, thread Γ_i is an element of the enabling list. Since the newly enabled thread is both an element of the enabling list and an ancestor of Γ_m^j, the claim holds.

5. To complete the proof of the proposition, we finally consider the case where some other processor steals a thread from processor p at time step t_{j-1}.

   Recall that the stolen thread is either an element of the enabling list (the one of the smallest timestamp among the elements of the enabling list) or $\Gamma^{j-1}_{m_{j-1}}$. We proceed by case analysis on the stolen thread.

   Assume first that the stolen thread is an element of the enabling list. Then, by induction hypothesis all claims hold.

   Assume now that the stolen thread is the one that pointer $\mathtt{uptr}_p$ points. Then, $m_j = m_{j-1} - 1$ and $n_j = n_{j-1} - 1$. Moreover, for each $i = 1, \ldots, m_j$, $\Gamma^j_i = \Gamma^{j-1}_i$. Thus, (1) holds by induction hypothesis (1). Clearly, for each $i = m_j + 1, \ldots, n_j$, $\Gamma^j_i = \Gamma^{j-1}_{i+1}$. By induction hypothesis (2), for all $i = m_{j-1}+1, \ldots, n_{j-1}$, thread $\Gamma^{j-1}_i = \Gamma^j_{i+1}$ is an element of the enabling list. Thus, for all $i = m_{j-1}, \ldots, n_{j-1} - 1$, thread Γ^j_i is an element of the enabling list. However, $m_{j-1} = m_j + 1$ and $n_j = n_{j-1} - 1$, so that for each $i = m_j, \ldots, n_j$, thread Γ^j_i is an element of the enabling list, as needed.

The proof of the proposition is now complete. ■

An immediate consequence of Proposition C.2 is that at every time step of the execution of any strict multithreaded computation by algorithm SRWS, pointer $\mathtt{uptr}_p$ of each processor p is either `null` or it points to the topmost thread Γ in p's ready list, such that Γ has not been worked on since it last spawned on processor p.

Corollary C.3 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that $p \in [P]$ is any processor. Then, pointer $\mathtt{uptr}_p$ points to the topmost thread in p's ready list that has not been worked on since it last spawned on processor p, or it is* `null` *if no such thread exists in p's ready list.*

Next proposition establishes that a thread can be stolen from some processor's ready list only after all its ancestor threads of smaller timestamp in p's ready list have been stolen.

Proposition C.4 *Let G be any multithreaded computation and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step t during the course of an execution of G by some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Let Γ, Γ' be threads in p's ready list such that thread Γ is an ancestor of thread Γ' and Γ has a smaller timestamp than Γ'. Then, thread Γ' can not be stolen from p's ready list, unless thread Γ is stolen.*

Proof: By case analysis on Γ.

Assume first that Γ is an element of the enabling list at time step t. Recall that when a processor steals a thread from p's ready list it steals the thread of smallest timestamp among the one pointed by pointer $\mathtt{uptr}_p$ and those in p's enabling list. Since thread Γ has smaller timestamp than thread Γ', independently of whether Γ' is an element of the enabling list or not, thread Γ must be stolen before thread Γ', as needed.

Assume now that Γ is not an element of p's enabling list. Since Γ is an ancestor of Γ', Lemma C.1 implies that Γ' resides in a position of p's ready list closer to the bottomost end of p's ready list than Γ. Thus, Proposition C.2 implies that Γ' is not an element of p's enabling list at time step t. By both the way stealing is performed and pointer `uptr` is updated, $\texttt{uptr}_p$ "meets" thread Γ before thread Γ', so that thread Γ' can not be stolen, unless thread Γ is stolen, as needed. ∎

The following proposition presents fundamental properties of the structure of the ready list of any processor. These properties are very important for proving the performance bounds of our algorithm.

Proposition C.5 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that an arbitrary processor $p \in [P]$ is working on a thread at some given time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Let Γ_0 be the thread that p is working on, let n be the number of threads in p's ready list and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the threads in p's ready list ordered from bottom to top, so that Γ_1 is bottomost and Γ_n is topmost. If $n > h+1$, then the threads in p's ready list satisfy the following properties:*

(1) *for all $i = 1, \ldots, n-h$, thread Γ_{i-1} is a child of thread Γ_i in the spawn tree;*

(2) *for all $i = 1, \ldots, n-h-1$, thread Γ_i has not been worked on since it spawned Γ_{i-1};*

(3) *if $i \in \{h+1, \ldots, n\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l has larger timestamp than thread Γ_i.*

Proof: By induction on the execution time. Let $t_0, t_1, \ldots, t_T$ be the time steps of the execution and let $p \in [P]$ be any arbitrary processor. For simplicity of proof we use the following notation. For each $j \in [T]$, let Γ_0^j be the thread that processor p works on at time step t_j. Notice that Γ_0^j might be `null`, if the processor does not have work to do at time step t_j (e.g., if it tries to steal some thread at t_j). Moreover, assume that the ready list of processor p contains $n_j \geq 0$ ready threads at time step t_j, which we denote by $\Gamma_1^j, \ldots, \Gamma_{n_j}^j$, ordered from bottom to top, so that Γ_1^j is bottomost and $\Gamma_{n_j}^j$ is topmost. Furthermore, for any integer $j \in [T]$, and any arbitrary thread Γ of p's ready list at time step t_j, we denote by $\mathit{timestamp}_j(\Gamma)$, the timestamp of thread Γ at time step t_j.

Basis Case

For the basis case where $t = t_0$ the claims hold vacously, since the only ready thread is the root thread, which resides in the ready list of some processor, while all other ready lists are empty.

Induction Hypothesis

Assume now that the claims hold for some time step t_{j-1} such that $j \in [T]$; that is, if we have $n_{j-1} > h+1$, then the threads in p's ready list satisfy the following properties:

(1) for all $i = 1, \ldots, n_{j-1} - h$, thread Γ_{i-1}^{j-1} is a child of thread Γ_i^{j-1} in the spawn tree;

(2) for all $i = 1, \ldots, n_{j-1} - h - 1$, thread Γ_i^{j-1} has not been worked on since it spawned Γ_{i-1}^{j-1}.

(3) if $i \in \{h+1, \ldots, n_{j-1}\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l^{j-1} has larger timestamp than thread Γ_i^{j-1}.

Induction Step

We will prove that the claims hold for t_j; that is, if $n_j > h+1$ then the threads in p's ready list satisfy the following properties:

(1) for all $i = 1, \ldots, n_j - h$, thread Γ_{i-1}^{j} is a child of thread Γ_i^{j} in the spawn tree;

(2) for all $i = 1, \ldots, n_j - h - 1$, thread Γ_i^{j} has not been worked on since it spawned Γ_{i-1}^{j};

(3) if $i \in \{h+1, \ldots, n_j\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l^{j} has larger timestamp than thread Γ_i^{j}.

The following cases should be considered for the action taken by the task, let it be γ, that executes at time step t_{j-1}: (1) γ is a "simple" instruction, (2) γ spawns a new thread, (3) thread Γ_0^j stalls or dies, and (4) γ enables a stalled thread. Moreover, the case that another processor steals a thread from p's ready list should be considered. Thus, we proceed by case analysis.

1. We consider first the case where γ executes a "simple" instruction.

   Then, $n_j = n_{j-1}$, $\Gamma_0^j = \Gamma_0^{j-1}$ and for each $i \in [n_j]$, $\Gamma_i^j = \Gamma_i^{j-1}$, and $\mathit{timestamp}_{j-1}(\Gamma_i^j) = \mathit{timestamp}_j(\Gamma_i^j)$, so that induction hypothesis immediately implies that all claims hold.

2. We now consider the case where the executed task spawns a thread.

   Clearly, $n_j = n_{j-1} + 1$. According to the way the internal scheduler works, the newly created thread becomes the active thread that the processor works on, while the thread that performed the spawning is placed back in the bottomost position of the ready list. Thus, for each $i \in [n_j]$ we have that $\Gamma_i^j = \Gamma_{i-1}^{j-1}$.

   In case $n_{j-1} < h+1$, the claims hold vacously. Assume now that $n_{j-1} \geq h+1$, so that $n_j = n_{j-1} + 1 > h+1$. Induction hypothesis implies that: (1) for all $i = 1, \ldots, n_{j-1} - h$, thread $\Gamma_{i-1}^{j-1} = \Gamma_i^j$ is a child of thread $\Gamma_i^{j-1} = \Gamma_{i+1}^j$ in the spawn tree; (2) for all $i = 1, \ldots, n_{j-1} - h - 1$, thread Γ_i^{j-1} has not been worked on since it spawned Γ_{i-1}^{j-1}; (3) if $i \in \{h+1, \ldots, n_{j-1}\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l^{j-1} has larger timestamp than thread Γ_i^{j-1}. Thus, (1) for all $i = 2, \ldots, n_j - h$, thread Γ_{i-1}^j is a child of thread Γ_i^j in the spawn tree; (2) for all $i = 2, \ldots, n_j - h - 1$, thread Γ_i^j has not been worked on since it spawned Γ_{i-1}^j; (3) if $i \in \{h+1, \ldots, n_j\}$, for all $l = 1, \ldots, i-h-1$, thread Γ_l^j has larger timestamp than thread Γ_i^j.

Clearly, thread Γ_0^j is a child of thread $\Gamma_1^j = \Gamma_0^{j-1}$, while thread Γ_1^j has not been worked on since it spawned Γ_0^j because the spawn only just occured. Moreover, the newly created thread has the larger timestamp among all threads in p's ready list. Thus, all claims hold.

3. We continue to consider the case where thread Γ_0^{j-1} stalls or dies.

   If $n_{j-1} = 0$, the ready list is empty and the processor p commences work-stealing. Independently of whether it will acquire work or not all claims hold vacously. Otherwise, $n_{j-1} > 0$, so that the ready list is not empty. The internal scheduler pops the bottomost thread from the ready list and the processor starts work on it. In case that $n_j \le h+1$, the claims hold vacously. Consider finally the case where $n_j > h+1$. Then, we have $n_j = n_{j-1} - 1$ and $\Gamma_0^j = \Gamma_1^{j-1}$. Moreover, for all $i = 1, \ldots n_j = n_{j-1} - 1$, we have $\Gamma_i^j = \Gamma_{i+1}^{j-1}$. Induction hypothesis implies that, (1) for all $i = 2, \ldots, n_{j-1} - h$, thread $\Gamma_{i-1}^{j-1} = \Gamma_{i-2}^j$ is a child of thread $\Gamma_i^{j-1} = \Gamma_{i-1}^j$ in the spawn tree; (2) for all $i = 2, \ldots, n_{j-1} - h - 1$, thread $\Gamma_i^{j-1} = \Gamma_{i-1}^j$ has not been worked on since it spawned $\Gamma_{i-1}^{j-1} = \Gamma_{i-2}^j$; (3) if $i \in \{h+1, \ldots, n_{j-1}\}$, for all $l = 1, \ldots, i-h-1$, thread $\Gamma_l^{j-1} = \Gamma_{l-1}^j$ has larger timestamp than thread $\Gamma_i^{j-1} = \Gamma_{i-1}^j$. Thus, (1) for all $i = 1, \ldots, n_{j-1} - h - 1 = n_j - h$, thread Γ_{i-1}^j is a child of thread Γ_i^j in the spawn tree; (2) for all $i = 1, \ldots, n_{j-1} - h - 1 - 1 = n_j - h - 1$, thread Γ_i^j has not been worked on since it spawned Γ_{i-1}^j; (3) if $i \in \{h+1, \ldots, n_{j-1} - 1 = n_j\}$, for all $l = 0, \ldots, i-h-1$, thread Γ_l^j has larger timestamp than thread Γ_i^j, as needed.

4. We now consider the case that γ enables a stalled thread.

   Let Γ_e^j be the enabled thread. If $n_{j-1} < h+1$, $n_j = n_{j-1} + 1 \le h+1$, so that the claims hold vacously. Thus, the only interesting case is if $n_{j-1} \ge h+1$, so that $n_j > h+1$.

   We proceed to prove:

   Lemma C.6 *$e \le h$.*

   Proof: Clearly, $\gamma \in \Gamma_0^{j-1}$, for otherwise, task γ could not enable thread Γ_e^j at time step t_{j-1}. By the way the internal scheduler works, no descendant of Γ_0^{j-1} can be in p's ready list at t_{j-1} and thus at t_j, for otherwise this descendant must have been executed at time step t_{j-1} instead of Γ_0^{j-1}. Clearly, $\Gamma_0^{j-1} = \Gamma_0^j$. By definition of dependency height, thread Γ_e^j is an l-ancestor of Γ_0^{j-1} with $l \le h$, so that $e \le h$, as needed. ■

   We proceed by case analysis. In case $e = n_j$, Lemma C.6 implies that at most h threads exist in p's ready list at t_j. Thus, $n_j < h+1$, so that all claims hold vacously.

   hence, it must be the case $e < n_j$. Lemma C.6 implies that $e \le h$. Since $n_j > h+1$, let $i \in \{h+1, \ldots, n_j\}$ be any integer. Assume that thread Γ_i^j is placed in p's ready list at time step t_{r_i}, $r_i < j$, and it is not worked on between time steps t_{r_i} and t_j. We start by proving that no l-descendant of thread Γ_i^j such that $l > h$ resides in p's ready list at time step t_{r_i}.

Lemma C.7 *Assume that Γ is an l-descendant of Γ_i^j such that $l > h$. Then, Γ does not reside in p's ready list at time step t_{r_i}.*

Proof: There are three cases to consider for the way that Γ_i^j is placed in p's ready list at time step t_{r_i}: (1) either Γ_i^j is spawned on processor p, or (2) p steals Γ_i^j from the ready list of some other processor, or (3) Γ_i^j is enabled by some of its descendants executing on processor p. Thus, we proceed by case analysis on Γ_i^j.

Assume first that Γ_i^j is spawned on processor p at time step t_{r_i}. Since Γ is a descendant of Γ_i^j, Γ is spawned after Γ_i^j, that is after time step t_{r_i}. Thus, Γ does not reside in p's ready list at time step t_{r_i}.

Assume next that p steals Γ_i^j from the ready list of some other procesor at time step t_{r_i}. Recall that a processor commences work-stealing only if its ready list is empty, so Γ does not reside in p's ready list at time step t_{r_i}.

Assume now that some thread Γ', which is a descendant of Γ_i^j, enables and places thread Γ_i^j in p's ready list at time step t_{r_i}. Since Γ is an l-descendant of Γ_i^j with $l > h$, definition of dependency height implies that Γ' is an ancestor of Γ. Since the internal scheduler always works on the bottomost thread of p's ready list, thread Γ does not exist in p's ready list at time step t_{r_i}, for otherwise, thread Γ' is not executed at time step t_{r_i} and thus the enabling of Γ_i^j can not occur at t_{r_i}, which contradicts the assumption than Γ_i^j is enabled by thread Γ' at time step t_{r_i}.

This completes the proof of the lemma. ■

We continue to prove that p does not commence work-stealing from time step t_{r_i} till time step t_j.

Lemma C.8 *Processor p does not commence work-stealing from time step t_{r_i} till time step t_j.*

Proof: Let t be any time step such that $t_{r_i} < t < t_j$. Clearly, thread Γ_i^j resides in p's ready list at time step t. Since p commences work-stealing only if its ready list is empty, p does not commence work-stealing at time step t. Thus, p does not steal any thread between time steps t_{r_i} and t_j. ■

We continue to prove that no l-descendant of Γ_i^j such that $l > h$ can be placed in p's ready list by enabling at some time step between t_{r_i} and t_j.

Lemma C.9 *Assume that Γ is an l-descendant of Γ_i^j such that $l > h$. Then, Γ can not be placed in p's ready list by enabling at some time step t such that $t_{r_i} \leq t \leq t_j$.*

Proof: By induction on the execution time. For the basis case $t = t_{r_i}$. Lemma C.7 implies that Γ and all its descendants do not reside in p's ready list at time step t_{r_i}. Thus, no enabling of these threads can occur at time step t_{r_i}, as needed.

Assume inductively that for any time step t such that $t_{r_i} < t \leq t_j$, no l-descendant of Γ^j_i, $l > h$, has been placed in p's ready list by enabling at any time step t' such that $t_{r_i} \leq t' < t$.

We prove that no l-descendant of Γ^j_i, $l > h$, is placed in p's ready list by enabling at time step t.

Assume that there exist m l-descendants, $l > h$ of Γ^j_i in p's ready list at time step t. For simplicity of proof we denote them by $\Gamma^t_{i-h-1}, \ldots, \Gamma^t_{i-h-m}$. By induction hypothesis none of these threads has been placed in p's ready list by enabling till time step t. Lemma C.8 implies that none of these threads has been placed in p's ready list by stealing. Thus, all these threads have been placed by spawning after time step t_{r_i}, so that all these threads have larger timestamps than thread Γ^j_i. Proposition C.4 implies that no such thread can be stolen from p's ready list, unless Γ^j_i is stolen. Since Γ^j_i is not stolen till time step t_j, no such thread is stolen either till time step t_j. Thus, for all $i = i-h-m+1, \ldots, i-h-1$, thread Γ^t_{i-1} is a child of thread Γ^t_i, so that none of Γ^t_i is missing from p's ready list and thus, none of them can be enabled at time step t. Moreover, no descendant of Γ^j_{i-h-m} resides in p's ready list, so that no descendant of Γ^t_{i-h-m} can be enabled at time step t.

This completes the proof of the lemma. ■

We now prove (1); that is, for each $i = 1, \ldots, n_j - h = n_{j-1} + 1 - h$, thread Γ^j_{i-1} is a child of thread Γ^j_i in the spawn tree. Induction hypothesis (1) implies that for each $i = 1, \ldots, n_{j-1} - h = n_j - h - 1$, thread Γ^{j-1}_{i-1} is a child of thread Γ^{j-1}_i at the spawn tree. Thus, the enabled thread should be one of the $(h+1)$ topmost threads in the ready list of processor p at time step t_j. Since $n_j = n_{j-1} + 1$ and the enabled thread is one of the (h+1) topmost, proving that thread $\Gamma^j_{n_j-h}$ is the parent of thread $\Gamma^j_{n_j-h-1}$ completes the proof of (1).

Assume, by the way of contradiction, that $\Gamma^j_{n_j-h}$ is not the parent of thread $\Gamma^j_{n_j-h-1}$. Let thread Γ be the parent thread of $\Gamma^j_{n_j-h-1}$. Lemma C.1 implies that for each $i = 1, \ldots, h+1$, thread $\Gamma^j_{n_j-i+1}$ is an ancestor of thread $\Gamma^j_{n_j-i}$. Thus, there exist $(h+1)$ ancestors of thread $\Gamma^j_{n_j-h-1}$ in positions of p's ready list closer to the top endpoint than that of thread $\Gamma^j_{n_j-h-1}$, none of which is Γ. Since $\Gamma^j_{n_j-h}$ is not the parent of thread $\Gamma^j_{n_j-h-1}$, this implies that thread $\Gamma^j_{n_j}$ is an l-ancestor of thread $\Gamma^j_{n_j-h-1}$, for some integer $l > h+1$. Assume that thread $\Gamma^j_{n_j}$ is placed in p's ready list at time step t_r, and it is not got stolen between time steps t_r and t_j.

Lemma C.7 implies that thread $\Gamma^j_{n_j-h-1}$ does not reside in p's ready list at time step t_r. However, thread $\Gamma^j_{n_j-h-1}$ resides in p's ready list at time step t_j. Lemmas C.8 and C.9 imply that $\Gamma^j_{n_j-h-1}$ is not placed in p's ready list by stealing or enabling. Thus, $\Gamma^j_{n_j-h-1}$

is spawned on processor p at some time step t such that $t_r < t < t_j$. Since $\Gamma^j_{n_j-h-1}$ is spawned on processor p, its parent Γ also exists in p's ready list at time step t (since it performs the spawning).

Since $\Gamma^j_{n_j-h-1}$ is an l-descendant of $\Gamma^j_{n_j}$ such that $l > h+1$, thread Γ must be an l-descendant of $\Gamma^j_{n_j}$ such that $l > h$. Thus, Lemma C.7 implies that Γ is not in p's ready list at time step t_r, so that Γ is placed in p's ready list at some time step after t_r. Hence, Γ has larger timestamp than $\Gamma^j_{n_j}$. Since $\Gamma^j_{n_j}$ is not stolen again after time step t_r and before time step t_j, Proposition C.4 implies that Γ can not be stolen either between these two time steps. However, Γ is not in p's ready list at time step t_j. A contradiction.

This completes the proof of (1), in case γ enables a thread at time step t_{j-1}.

We proceed to prove (2).

Lemma C.10 *For all $i = 1, \ldots, n_j - h - 1$, thread Γ^j_i has not been worked on since it spawned Γ^j_{i-1}.*

Proof: Induction hypothesis (1) implies that for each $i = 1, \ldots, n_{j-1} - h = n_j - h - 1$, thread Γ^{j-1}_{i-1} is a child of thread Γ^{j-1}_i at the spawn tree. Thus, thread Γ^j_e should be one of the $(h+1)$ topmost threads in the ready list of processor p at time step t_j, so that $\Gamma^j_e \neq \Gamma^j_{n_j-h-1}$.

Consider thread $\Gamma^j_{n_j}$ and let t_r be a time step such that thread $\Gamma^j_{n_j}$ is placed in p's ready list at time step t_r, and it is not got stolen between time steps t_r and t_j.

Lemma C.1 implies that $\Gamma^j_{n_j-h-1}$ is an l-descendant of $\Gamma^j_{n_j}$ such that $l > h$. Lemma C.7 implies that thread $\Gamma^j_{n_j-h-1}$ does not exist in p's ready list at time step t_r. Lemma C.8 implies that processor p does not steal thread $\Gamma^j_{n_j-h-1}$ from the ready list of some other processor at any time step between t_r and t_j. Lemma C.9 implies that thread $\Gamma^j_{n_j-h-1}$ can not be enabled by any of its descendants at any time step between t_r and t_j. Thus, the only way for $\Gamma^j_{n_j-h-1}$ to exist in p's ready list at time step t_j is if it is spawned on processor p at some time step t such that $t_r < t < t_j$. In this case, thread $\Gamma^j_{n_j-h-1}$ has larger timestamp than $\Gamma^j_{n_j}$. Since $\Gamma^j_{n_j}$ is not stolen from p's ready list for all time steps between t_r and t_j, Lemma C.4 implies that thread $\Gamma^j_{n_j-h-1}$ is not stolen either. Thus, thread $\Gamma^j_{n_j-h-1}$ has not been worked on, since it spawned thread $\Gamma^j_{n_j-h-2}$. Induction hypothesis (2) implies that for each $i \in n_j - h - 2, \ldots 1$, thread Γ_i has not been worked on since it spawned Γ_{i-1}.

The proof of the lemma is now complete. ■

At this point we have completed the proof of (2), in case γ enables some thread at time step t_{j-1}.

We finally prove (3). Let $i \in \{h+1, \ldots, n_j\}$ be any integer. Assume that thread Γ_i^j is placed in p's ready list at time step t_{r_i}, $r_i < j$, and it not got stolen between time steps t_{r_i} and t_j.

Lemma C.7 implies that for all $l = 0, \ldots, i-h-1$, thread Γ_l^j does not exist in p's ready list at time step t_{r_i}. Lemma C.8 implies that processor p does not steal thread Γ_l^j from the ready list of some other processor at any time step between t_r and t_j. Lemma C.9 implies that thread Γ_l^j can not be enabled by any of its descendants at any time step between t_{r_i} and t_j. Thus, the only way for Γ_l^j to exist in p's ready list at time step t_j is if it is spawned on processor p at some time step t, $t_{r_i} < t < t_j$. Thus, thread Γ_l^j has larger timestamp than Γ_i^j at time step t.

Lemma C.10 implies that for each $q = 1, \ldots, n_j - h - 1$, thread Γ_q^j has not been worked on since it spawned Γ_{q-1}^j. Since $i \leq n_j$, $l \leq n_j - h - 1$, this implies that thread Γ_l^j has not been worked on since it spawned thread Γ_{l-1}^j, so that its timestamp does not change till time step t_j. Thus, for each $l = 0, \ldots, i-h-1$, thread Γ_l^j has larger timestamp than thread Γ_i^j, as needed by (3).

5. To complete the proof of the proposition, we finally consider the case where some other processor steals a thread from processor p at time step t_{j-1}.

   Recall that a processor steals the thread of smallest timestamp among the one pointed by pointer $\texttt{uptr}_p$ and the elements of the enabling list. By induction hypothesis (2), for all $i = 1, \ldots, n_{j-1} - h - 1$, thread Γ_i^{j-1} has not been worked on since it spawned Γ_{i-1}^{j-1}. Corollary C.3 implies that pointer $\texttt{uptr}_p$ points either to $\Gamma_{n_{j-1}-h-1}^{j-1}$ or to some of its ancestors. We proceed by case analysis on the thread pointed by $\texttt{uptr}_p$ at time step t_{j-1}.

   Assume first that $\texttt{uptr}_p$ points to $\Gamma_{n_{j-1}-h-1}^{j-1}$ at t_{j-1}. Proposition C.2(2) implies that $\Gamma_{n_{j-1}}^{j-1}$ is an element of the enabling list. Induction hypothesis (3) implies that thread $\Gamma_{n_{j-1}-h-1}^{j-1}$ has a larger timestamp than $\Gamma_{n_{j-1}}^{j-1}$. Thus, by the way stealing is performed, $\Gamma_{n_{j-1}-h-1}^{j-1}$ can not be stolen at time step t_{j-1}. Clearly, the stolen thread is the element of the enabling list with the smallest timestamp. However, since pointer $\texttt{uptr}_p$ points to $\Gamma_{n_{j-1}-h-1}^{j-1}$, Proposition C.2(2) implies that there exists at most $(h+1)$ thread in p's enabling list at time step t_{j-1}. Thus, one of the $(h+1)$ topmost threads in p's ready list is stolen. Cleary, $n_j = n_{j-1} - 1$, so that induction hypothesis implies that all claims hold.

   Assume now that pointer $\texttt{uptr}_p$ points to some ancestor Γ' of thread $\Gamma_{n_{j-1}-h-1}^{j-1}$ at t_{j-1}. Proposition C.2(2) implies that all elements of the enabling list are ancestors of Γ'. Thus, the stolen thread is again one of the $(h+1)$ topmost threads, so that induction hypothesis implies that all claims hold.

The proof of Proposition C.5 is now complete. ■

An immediate consequence of Propositions C.2 and C.5 is that the enabling list of any processor can not contain more than $(h+1)$ elements.

Corollary C.11 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then, for any timestep t during the course of the execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$, the enabling list of processor p contains at most $(h+1)$ elements.*

C.1.2 Properties of Critical Tasks

In this section we prove that during the execution of a multithreaded computation by algorithm SRWS, a critical task, residing in the ready list of some processor p, is either an element of p's enabling list or pointer $\mathtt{uptr}_p$ points to it. This implies that a critical task is always one of the $(h+2)$ topmost threads in the ready list of any processor p. Moreover, we prove that a critical task is executed after at most $(2h+1)$ steal attempts that target processor p. We finally conclude that a critical task is unlikely to remain critical across a modest number of steal attempts.

Proposition C.12 *Consider a strict multithreaded computation G and let $P > 1$ be any integer. At every time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$, each critical task must be the ready task of a thread that either is an element of the enabling list of some processor p or pointer $\mathtt{uptr}_p$ points to it.*

Proof: Consider any time step during the course of the execution of G according to the execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Let $\gamma \in \Gamma$ be any critical task. Since γ is critical it is ready and thus Γ resides in some processor's ready list; let it be processor p.

Assume by the way of contradiction that Γ is neither an element of p's enabling list nor the thread that $\mathtt{uptr}_p$ points to. Let $\mathtt{uptr}_p$ point to some thread Γ'. Proposition C.2 implies that Γ' is not an element of p's enabling list. Since both Γ and Γ' are not elements of the enabling list, Corollary C.2 implies that Γ' is an ancestor of Γ. Let γ' be the task of Γ' that spawns the subtree that Γ resides and let γ'' be the "successor" of task γ' in G'. Observation B.2 implies that there exists a path from γ'' to γ. Since γ is critical γ'' must have been executed. However, Proposition C.2(2) implies that γ'' has not been executed. A contradiction.

Thus, Γ is either an element of p's enabling list or pointer $\mathtt{uptr}_p$ points to Γ, which completes the proof of the proposition. ■

Next claim is an immediate consequence of Proposition C.12 and Corollary C.11.

Corollary C.13 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. At every time step, during the execution of any schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$, each critical task must be the ready task of a thread that is one of the $(h+2)$ topmost threads in its processor's ready list.*

We continue to prove that during the execution of any strict multithreaded computation by Algorithm SRWS on a parallel computer with P processors, any critical task executes after at most $(2h+1)$ steal requests have been serviced to the ready list of the processor that the thread containing this task resides.

Proposition C.14 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Assume that γ is the critical task of a thread Γ residing in the ready list of processor $p \in [P]$, at some time step during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Then, after at most $(2h+1)$ steal requests have been serviced on processor p, task γ executes.*

Proof: Let t_{s_0} be the time step that γ becomes critical. Let $t_{s_1}, t_{s_2}, \ldots, t_{s_{2h+1}}$ be the time steps that each one of the $1, 2, \ldots, 2h+1$ steal requests are serviced, respectively. Denote by t_γ the time step that task γ executes.

Clearly, there exists at least one descendant of Γ in p's ready list for all time steps t such that $t_{s_0} \leq t < t_\gamma$, for otherwise, processor p works on thread Γ at time step t, so that task γ is executed at $t < t_\gamma$.

We proceed by case analysis on Γ.

1. Assume first that thread Γ is not an element of the enabling list at t_{s_0}.

   Since γ is critical, Proposition C.12 implies that pointer $\mathtt{uptr}_p$ points to Γ at t_{s_0}. By Corollary C.11, there exist at most $(h+1)$ elements in p's enabling list at t_{s_0}. Clearly, all threads that are enabled and placed in p's ready list after time step t_{s_0} have larger timestamp than any other thread residing in p's ready list at t_{s_0}, so that all these threads have larger timestamp than Γ. Thus, after $(h+1)$ steal requests have been serviced on processor p, either Γ has been stolen or all elements of p's enabling list at t_{s_0} have been stolen. In case Γ has been stolen by time step $t_{s_{h+1}}$, task γ is executed after at most $(h+1) < (2h+1)$ steal requests have been serviced on processor p, as needed. Assume now that task γ is still in p's ready list at time step $t_{s_{h+1}}$. By the way $\mathtt{uptr}_p$ is updated, $\mathtt{uptr}_p$ still points to thread Γ. Thus, either Γ is stolen at time step $t_{s_{h+2}}$ or processor p executes γ at some time step before $t_{s_{h+2}}$. In both cases the claim holds.

2. Assume now that Γ is an element of the enabling list at t_{s_0}.

   Corollary C.11 implies that p's enabling list contains at most h elements other than Γ at t_{s_0}. Clearly, all threads that are enabled and placed in p's ready list after time step t_{s_0} have larger timestamp than any other thread residing in p's ready list at t_{s_0}, so that all these threads have larger timestamp than Γ. Recall that a thief processor always steals the thread of smallest timestamp among the one pointed by pointer $\mathtt{uptr}_p$ and those of p's enabling list. Proposition C.5 (3) implies that at most h descendants of Γ in p's ready list at t_{s_0} may have smaller timestamp than Γ. Moreover, at most h ancestors of Γ, namely, all h elements different than Γ in p's enabling list at t_{s_0}, may also have a smaller timestamp than Γ. Thus, after at most $2h$ steal requests have been servirced on processor p, either γ has been executed, or all threads with smaller timestamp than Γ in p's ready list have been executed. In case Γ has been stolen by time step $t_{s_{2h}}$, task γ is executed after at most $2h < (2h+1)$ steal requests have been serviced on processor p, as needed. Assume now that thread Γ is still in p's ready list at time step $t_{s_{2h}}$. Then, Γ is the thread of smaller timestamp in p's enabling list, so that either γ is executed at time step $t_{s_{2h+1}}$, or processor p executes γ at some time step t, such that $t_{s_{2h}} < t < t_{s_{2h+1}}$. In both cases

γ executes after at most $(2h+1)$ steal attempts have been serviced on processor p, as needed.

The proof of the proposition is now complete. ■

We next prove that during the execution of any strict multithreaded computation by Algorithm SRWS on a P-processor parallel computer, the probability that $r > 1$ rounds of steal attempts occur while any particular task is critical is exponentially decreasing with hr.

Proposition C.15 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then, for any task γ and any integer $r > 1$, the probability that r rounds occur during the course of an execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G,P)$, while γ is critical is at most e^{-2hr}.*

Proof: Let t_γ denote the first time step at which task γ of some thread Γ is critical. Since γ is critical, thread Γ is ready and thus it resides in the ready list of some processor, let it be p. Suppose r steal attempt rounds occur while γ is critical. Clearly, at least $6hrP$ steal attempts occur during the r rounds. Denote by t_s the time step that the last of these steal attempts is initiated and let t_e be the time step that γ executes. Clearly, it must be that $t_s < t_e$. Consider now the time step t_{s-2h}; that is, t_{s-2h} is the time step $2h$ time steps before the time step that the last steal attempt is initiated. Since, there are $(2h+1)$ execution time steps between t_{s-2h} and t_s (both inclusive), and at most P steal attempts can be initiated per time step, at least $6hrP - 2hP = (6r-2)hP = (3r-1)2hP$ of the $6hrP$ steal attempts must be initiated at a time step before or at t_{s-2h}. Proposition C.14 immediately implies that at most $2h$ of these steal attempts, initiated $(2h+1)$ time steps before the execution of γ, can choose processor p as their target, for otherwise, γ would be executed at or before $t_s < t_e$.

Thus, we must have $(3r-1)2hP$ steal attempts, each initiated at a time step t such that $t_\gamma \leq t \leq t_{s-2h}$ and at most $2h$ of which target processor p. The probability that at most $2h$ steal attempts choose processor p as their target processor is the following:

$$
\begin{aligned}
& \sum_{i=0}^{2h} \binom{(3r-1)2hP}{i} \left(\frac{1}{P}\right)^i \left(1-\frac{1}{P}\right)^{(3r-1)2hP-i} \\
\leq\; & \binom{(3r-1)2hP}{2h} \left(\frac{1}{P}\right)^{2h} \left(1-\frac{1}{P}\right)^{(3r-1)2hP-2h} \frac{[(3r-1)2hP-2h+1]1/P}{[(3r-1)2hP]1/P-2h} \\
& \text{(by Lemma A.2)}\,.
\end{aligned}
$$

Notice that condition $2h < ((3r-1)2hP)+1)1/P$ is always true for $r > 1$, since:

$$
\begin{aligned}
((3r-1)2hP+1)1/P &= (3r-1)2h + 1/P \\
&= 6hr - 2h + 1/P \\
&\geq 4hr + 1/P \\
&> 4hr \\
&> 2h\,,
\end{aligned}
$$

whenever $r > 1$. Furthermore, for any integer $r > 1$, and any integer $h > 0$,

$$\begin{aligned}
& \frac{[(3r-1)2hP - 2h + 1]1/P}{[(3r-1)2hP]1/P - 2h} \\
= \; & \frac{(3r-1)2h - 2h/P + 1/P}{(3r-1)2h - 2h} \\
= \; & \frac{3r - 1 - 1/P + 1/(2hP)}{3r-2} \\
= \; & \frac{3r - 1 - (2h-1)/(2hP)}{3r-2} \\
= \; & \frac{3r-1}{3r-2} - \frac{2h-1}{2hP(3r-2)}\,.
\end{aligned}$$

Clearly,

$$0 < \frac{2h-1}{2hP(3r-2)} = \frac{2h-1}{2h}\,\frac{1}{(3r-2)P} < 1\,,$$

whenever $r > 1$, $h > 0$, while

$$1 < \frac{3r-1}{3r-2} < 2\,.$$

Thus,

$$0 < \frac{[(3r-1)2hP - 2h + 1]1/P}{[(3r-1)2hP]1/P - 2h} < 2\,,$$

so that

$$\begin{aligned}
& \sum_{i=0}^{2h} \binom{(3r-1)2hP}{i} \left(\frac{1}{P}\right)^{i} \left(1 - \frac{1}{P}\right)^{(3r-1)2hP-i} \\
\leq \; & 2 \binom{(3r-1)2hP}{2h} \left(\frac{1}{P}\right)^{2h} \left(1 - \frac{1}{P}\right)^{(3r-1)2hP-2h} \\
\leq \; & 2 \binom{(3r-1)2hP}{2h} \left(\frac{1}{P}\right)^{2h} e^{-(3r-1)2h+2h/P} \\
= \; & 2 \cdot \frac{[(3r-1)2hP]!}{(2h)!\,[(3r-1)2hP - 2h]!} \left(\frac{1}{P}\right)^{2h} e^{-(3r-1-1/P)2h} \\
\leq \; & 2 \cdot \frac{[(3r-1)2hP - 2h + 1] \ldots [(3r-1)2hP]}{(2h)!} \left(\frac{1}{P}\right)^{2h} e^{-(3r-2)2h} \\
\leq \; & 2\,\frac{[(3r-1)2hP]^{2h}}{(2h)!} \left(\frac{1}{P}\right)^{2h} e^{-(3r-2)2h} \\
= \; & 2\,\frac{[(3r-1)2hP]^{2h}}{(2h)^{2h}\,e^{-2h}}\,\frac{1}{\sqrt{4\pi h}} \left(\frac{1}{P}\right)^{2h} e^{-(3r-2)2h} \\
\approx \; & \text{(by Stirling Formula)} \\
= \; & 2 \left(\frac{(3r-1)2hP}{2hP}\right)^{2h} \frac{1}{\sqrt{4\pi h}}\, e^{-(3r-2)2h+2h}
\end{aligned}$$

$$
\begin{aligned}
&= 2\,(3r-1)^{2h}\,\frac{1}{\sqrt{4\pi h}}\,e^{-(3r-3)2h} \\
&\le 2\,(3r-1)^{2h}\,e^{-3(r-1)2h} \\
&\le e^{-2rh},
\end{aligned}
$$

whenever $r > 1$, as needed. ■

C.1.3 Properties of Delay Sequences

In this section we bound the probability that a particular delay sequence occurs during the execution of a strict multithreaded computation by Algorithm SRWS. More specifically we prove that for any strict multithreaded computation G, the probability for a particular delay sequence $\mathcal{D}(G)$ to occur is at most $e^{-2h(R-2L)}$.

Proposition C.16 *Consider any multithreaded computation G and let $P > 1$ be any integer. Assume that $\mathcal{D}(G) = \langle U(G), R, \Pi \rangle$ is any arbitrary delay sequence such that $U(G)$ is a critical path of length L. Then, the following holds for the probability that $\mathcal{D}(G)$ occurs during the execution of G according to some execution schedule $\mathcal{X}_{\mathsf{SRWS}}(G, P)$:*

$$\mathbf{P}[\mathcal{D}(G) occurs] \le e^{-2h(R-2L)}.$$

Proof: The delay sequence $\mathcal{D}(G)$ occurs if for each $i = 1, \ldots, L$, all π_i rounds of steal attempts occur while task γ_i is critical. Proposition C.15 implies that the probability of at least π_i rounds occuring while a given task is critical is at most $e^{-2h\pi_i}$, provided that $\pi_i > 1$. If $\pi_i \le 1$ we upper bound this probability by 1. Moreover, the targets of the steal attempts initiated at the π_i round are chosen independently from the targets chosen in other rounds. Thus, we have:

$$
\begin{aligned}
& \mathbf{P}[\mathcal{D}(G) = \langle U(G), R, \Pi \rangle \text{ occurs}] \\
&= \Pi_{\substack{1 \le i \le L \\ \pi_i > 1}} \mathbf{P}[\pi_i \text{ rounds occur while } \gamma_i \text{ is critical}] \\
&\le \Pi_{\substack{1 \le i \le L \\ \pi_i > 1}} e^{-2h\pi_i} \\
&= \mathsf{Exp}\left[-2h \sum_{\substack{1 \le i \le L \\ \pi_i > 1}} \pi_i\right] \\
&= \mathsf{Exp}\left[-2h \left(\sum_{1 \le i \le L} \pi_i - \sum_{\substack{1 \le i \le L \\ \pi_i \le 1}} \pi_i\right)\right] \\
&\le e^{-2h(R-L-L)} \\
& \text{(by Observation B.3)} \\
&= e^{-2h(R-2L)},
\end{aligned}
$$

which completes the proof. ■

D Complexity of the Algorithm

D.1 Space Complexity

Theorem D.1 *Consider any strict multithreaded computation G and let $P > 0$ be any integer. Then,*

$$S_{\mathsf{SRWS}}(G, P) \quad \leq \quad S_1 P\,.$$

Proof: We first prove Lemma D.2, which establishes that any execution schedule produced by algorithm SRWS maintains the busy leaves property. Let $\mathcal{X}_{\mathsf{SRWS}}(G, P)$ be any arbitrary execution schedule produced by Algorithm SRWS for the execution of computation G on a P-processor parallel computer.

Lemma D.2 *$\mathcal{X}_{\mathsf{SRWS}}(G, P)$ maintains the busy-leaves property.*

Proof: Let t be any time step during the execution of G by $\mathcal{X}_{\mathsf{SRWS}}(G, P)$. Every leaf thread of the spawn subtree at time step t is ready, so that it is in the ready list of some processor. However, Lemma C.1 and Proposition C.5 imply that no leaf thread resides in some processor's ready list, while the processor works on some other thread. ■

The stated claim now follows, as an immediate consequence of Proposition B.1 and Lemma D.2. ■

D.2 Execution Time

Proposition D.3 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then,*

(1) *for any $\epsilon > 0$, with probability at least $1 - \epsilon$,*

$$L_{\mathsf{SRWS}}(G, P) \quad \in \quad O(P(hT_\infty + \log(1/\epsilon)))\,;$$

(2) $\mathcal{E}(L_{\mathsf{SRWS}}(G, P)) \in O(PhT_\infty)$.

Proof: Proposition B.6 implies that if at least $(6hP + P)R$ steal attempts occur, then some delay sequence $\mathcal{D}(G) = \langle U(G), R, \Pi\rangle$ occurs. By Lemma B.4 it follows that there exist at most $(d+1)^{2T_\infty} \binom{4T_\infty + R}{R}$ delay sequences, while Proposition C.16 implies that each one of these delay sequences occurs with probability at most $e^{-2h(R-2L)}$. By Observation B.1, the dag depth of G' is $2T_\infty$. Thus, there are at most $2T_\infty$ tasks in the delay sequence, so that $L \leq 2T_\infty$.

Hence, the probability that any delay sequence occurs is bounded by:

$$
\begin{aligned}
& (d+1)^{2T_\infty} \binom{4T_\infty + R}{R} e^{-2h(R-2L)} \\
\leq\ & (d+1)^{2T_\infty} \binom{4T_\infty + R}{R} e^{-2h(R-4T_\infty)} \\
& (\text{since } L \leq 2T_\infty) \\
\leq\ & (d+1)^{2T_\infty} e^{4T_\infty + R} e^{-2h(R-4T_\infty)} \\
& (\text{by Lemma A.1}) \\
=\ & (d+1)^{2T_\infty} e^{-2h(R-4T_\infty)+4T_\infty+R}
\end{aligned}
$$

which is at most ϵ for $R = \Theta(T_\infty + \log(1/\epsilon)/h)$. Thus, the probability that at least $(6hP+P)R = \Theta(P(hT_\infty + \log(1/\epsilon)))$ steal attempts occur is at most ϵ.

The expectation bound follows, because the tail of the distribution decreases exponentially.
■

We now continue to analyze the total delay of processors due to contention caused by stealing at ready lists. In our analysis, we use the combinatorial "balls and bins" game described in Section A.1.

We start by describing how the (P, M)-recycling game, presented in Section A.1, models the servicing of steal requests by Algorithm SRWS. Every processor owns a ball and a bin. If the ball is in the reservoir, it means that the owner processor of this ball is not making a steal attempt in the current step. Otherwise, the owner processor has made a steal attempt to the ready list of the bin's owner processor that the ball resides in, but the attempt has not yet been satisfied. When a ball is removed from a bin and returned to the reservoir, it means that the request has been serviced.

By using the (P, M)-recycling game to model the servicing of steal requests, Proposition A.3 immediately implies that the total amount of time spent by processors waiting for the service of their stealing requests is likely to be proportional to the total number M of stealing requests. However, the total number of stealing requests is bounded by Proposition D.3. Thus, by these two propositions we have:

Proposition D.4 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then,*

(1) *for any $\epsilon > 0$, with probability at least $1 - \epsilon$,*

$$W_{\mathsf{SRWS}}(G, P) \in O(P(hT_\infty + \log P + \log(1/\epsilon)))\,;$$

(2) $\mathcal{E}(W_{\mathsf{SRWS}}(G, P)) \in O(PhT_\infty)$.

Since the work of any scheduling algorithm Alg on G is $T_1(G)$, Propositions D.3 and D.4, and Lemma B.2 immediately imply:

Theorem D.5 *Consider any strict multithreaded computation G, and let $P \geq 1$ be any integer. Then,*

(1) *for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the following holds:*

$$T_{\mathsf{SRWS}}(G, P) \in O(T_1/P + hT_\infty + \log P + \log(1/\epsilon)) .$$

(2) $\mathcal{E}(T_{\mathsf{SRWS}}(G, P)) \in O(T_1/P + hT_\infty)$.

D.3 Communication Complexity

Theorem D.6 *Consider any strict multithreaded computation G and let $P > 1$ be any integer. Then,*

(1) *for any $\epsilon > 0$, with probability at least $1 - \epsilon$,*

$$C_{\mathsf{SRWS}}(G, P) \in O(P(hT_\infty + \log(1/\epsilon))(1 + n_d)S_{max}) ;$$

(2) $\mathcal{E}(C_{\mathsf{SRWS}}(G, P)) \in O(PhT_\infty(1 + n_d)S_{max})$.

Proof: Communication occurs when either a steal attempt takes place or a dependency edge from some thread Γ' to some thread Γ residing on another processor than the one of Γ' is resolved. We assume that at most a constant number of bytes need be communicated along a dependency edge to resolve the dependency. Moreover, for every multithreaded computation we assume that the "migration" of one of its threads from a processor to another incurs communication at most $S_{max}(G)$.

When a steal attempt occurs either a thread is stolen or the steal attempt targets a processor with empty ready list. In both cases, the communication incurred is at most S_{max}, while the number of steal attempts is bounded by Proposition D.3. When a thread Γ' resolves a dependency to another thread Γ not in the same processor's ready list, Γ is placed in the ready list that Γ' resides. This event can happen at most n_d times for each time the thread is stolen, so the communication cost incurred is at most $n_d S_{max}$ per stealing. Moreover, since each dependency accounts for at most a constant number of messages, the communication cost of resolving any dependency is amortized by the cost $n_d S_{max}$. Thus, Proposition D.3 immediately implies the stated claims. ∎