# Tight upper bounds on the synchronization costs for Work Stealing

Guilherme Rito and Hervé Paulino

g.rito@campus.fct.unl.pt
herve.paulino@fct.unl.pt

NOVA Laboratory for Computer Science and Informatics
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa
2829-516 Caparica, Portugal

**Abstract**

Work Stealing has been a very successful algorithm for scheduling parallel computations, and is known to achieve high performances even for computations exhibiting fine-grained parallelism. In Work Stealing, each processor owns a deque that it uses to keep track of its assigned work. To ensure proper load balancing, each processor's deque is accessible not only to its owner but also to other processors that may be searching for work. However, due to the concurrent nature of deques, it has been proved that even when processors operate locally on their deque, synchronization is required. Moreover, many studies have found that synchronization related overheads often account for a significant portion of the total execution time. For these reasons, many efforts have been carried out to reduce the synchronization costs of conventional Work Stealing algorithms. The most recent of these have been focusing on eliminating synchronization for local deque operations by making deques private. Although these strategies have shown promising empirical results, it is not known whether they effectively avoid synchronization nor if they enjoy the same (asymptotically optimal) guarantees of Work Stealing with concurrent deques.

In this paper we study yet another such variant of Work Stealing. Our analysis shows that this strategy allows not only to maintain Work Stealing's asymptotically optimal guarantees, but also to effectively reduce synchronization. We showcase the great advantage of these approaches by obtaining (tight) upper bounds on the synchronization overheads for our algorithm.

## 1 Introduction

Multithreading is a fundamental mechanism to leverage the parallel design of multiprocessor chips. However, the performance of multithreaded computations is tightly bound to the efficiency of the underlying thread scheduler, responsible for keeping processors busy.

For some time now, the Work Stealing algorithm is one of the most popular for scheduling multithreaded computations. In Work Stealing, each worker (usually referred to as *processor*) owns a double-ended queue (deque) of threads ready to execute. This deque is locally manipulated as a stack, similarly to a sequential execution: processors push and pop threads from the bottom side of their deque when, respectively, a new thread is spawned and the execution of the current thread concludes. Additionally, whenever a pop operation finds the local deque empty, the processor becomes a *thief* and starts targeting other processors — called its *victims* — uniformly at random, with the purpose of stealing a thread from the top of their deques.

As shown by Blumofe *et al.* in [11], Work Stealing is provably efficient for scheduling multithreaded computations. However, due to the concurrent nature of processors' deques, the use of appropriate synchronization mechanisms is required to ensure correctness [9]. Work Stealing schedulers, effectively employed in systems such as Cilk [10,25], X10 [15], Intel TBB [23], Wool [20], Java Fork/Join [24], Hood [13], Pasl [3], among others, require the use of such synchronization mechanisms every time a processor adds or retrieves a thread from a deque. Consequently, even when processors operate locally on their deques, they incur expensive synchronization overheads that, in most cases, are unnecessary.

The first provably efficient Work Stealing algorithm, proposed by Blumofe *et al.* [11], assumed that all the steal attempts targeting each deque were serialized, and only ensured the success of at most one such attempt per time step. The idea was materialized in the Cilk run-time system [10] via a blocking synchronization protocol named *THE*. Although Cilk is known to be extremely efficient, Frigo *et al.* found that the overheads introduced by the *THE* protocol could easily get to more than half of the total execution time [21]. Subsequent non-blocking versions of Work Stealing [8,12] mitigated part of these overheads by replacing the *THE* protocol with a non-blocking one that resorts to Compare-And-Swap (CAS) and Memory Fence (MFence) instructions. Even so, the synchronization costs imposed by these operations are not negligible. For instance, Morrison *et al.* tuned Cilk by removing a single MFence instruction (one that was executed whenever a processor tried to take work from its deque) and found that this single Memory Fence could account for as much as 25% of the total execution time [28]. Unfortunately, it has been proved, by Attiya *et al.* in [9], that it is impossible to eliminate all synchronization from the implementation of concurrent deques, while maintaining correctness. Indirectly, this result implied the impossibility of eliminating all synchronization from Work Stealing algorithms that use concurrent deques.

Various proposals have been made with the goal of eliminating synchronization for local deque accesses, by making deques partly or even entirely private [2,26,28,31,32]. Nevertheless, even though the use of private deques eliminates synchronization for local accesses, it introduces new overheads that are not present in Work Stealing with concurrent deques. Consequently, and even though steals are usually rare, efforts have been placed in the use of partially private deques, known as Split Deques, or SpDeques for short, to avoid the drawbacks of using entirely private deques [17,18,26,28,31,32]. Like private deques, SpDeques avoid synchronization for local accesses by keeping their bottom part private, accessible only to their owner. However, SpDeques do not introduce extra overheads since, as in Work Stealing with concurrent deques, idle processors can take work directly from other processors' SpDeques, without requiring their intervention. The earliest work to use SpDeques for reducing synchronization in Work Stealing was, perhaps, by Dinan *et al.* in [18]. In their paper, the authors showcased the practical advantages of using SpDeques in large scale distributed settings, by comparing the performance of their algorithm when using SpDeques and when using concurrent deques. The advantages of SpDeques were clear, and, since then, numerous studies have further explored their use, obtaining favorable empirical results [26, 28, 31, 32]. Nevertheless, it is still unknown whether the performance gains of using SpDeques can be reflected in theory, and whether Work Stealing with SpDeques enjoys the same asymptotically optimal guarantees of Work Stealing with concurrent deques.

To this extent, the contributions of this paper are:

- The presentation of yet another Work Stealing algorithm — baptized Low Cost Work Stealing, or LCWS for short — that uses SpDeques to avoid most synchronization overheads (Section 2). As we will discuss in Section 4, LCWS differs from the remainder Work Stealing algorithms that use SpDeques. These differences make it appropriate for scheduling generic computations, and are key for obtaining bounds on the synchronization costs and execution time.

- The analysis of LCWS, which shows that for a $P$-processor execution of a computation with total work $T_1$ and critical-path length (*i.e.* span) $T_\infty$, the expected total synchronization costs are at most $O\left(\left(C_{CAS} + C_{MFence}\right) PT_\infty\right)$, where $C_{CAS}$ and $C_{MFence}$ respectively denote the synchronization costs incurred by the execution of a CAS and MFence instructions (Section 3). As we discuss in Section 5, this bound is tight and implies that for several classes of computations the use of SpDeques allows to reduce synchronization almost exponentially when compared with conventional Work Stealing algorithms that use concurrent deques. Additionally, we prove that the expected execution time of LCWS is $O\left(\frac{T_1}{P} + T_\infty\right)$, and briefly explain why the expected total communication of LCWS is a constant factor away from the expected total communication of Work Stealing with concurrent deques, which in turn is asymptotically optimal [11, 34]. These results justify the folk wisdom that SpDeques allow to effectively reduce synchronization while maintaining the same theoretical guarantees of Work Stealing with concurrent deques.

## 2 Low Cost Work Stealing

Like in much previous work [1, 2, 4, 5, 7, 8, 11, 29, 30], we model computations as dags — where each node corresponds to an instruction and each edge denotes an ordering constraint between two instructions — and assume that computations have a single root, a single sink and that the out-degree of any node is at most two. The span and number of nodes of a dag are denoted, respectively, by $T_\infty$ and $T_1$. A node is *ready* if all its ancestors have been executed, and to ensure correction only ready nodes can be executed. When a becomes ready say that it was *enabled.* The assignment of a node $\mu$ to a processor $p$ means that $\mu$ will be the next node $p$ executes.

In LCWS, each processor owns a Lock Free SpDeque, instead of a typical concurrent deque. An SpDeque (illustrated in Figure 1) is simply a deque that is split into two parts: a private part and a public part. The public part lies in the top of the SpDeque whereas the private corresponds to the rest of the SpDeque. To avoid synchronization for local operations, only the owner of an SpDeque is allowed to access its private part. Furthermore, by default busy processors operate on the private part of their SpDeque, pushing and popping ready nodes as necessary. In fact, a busy processor only attempts to fetch work from the public part of its SpDeque if the private part is empty. In such situation, if the processor's attempt succeeds (*i.e.* if the public part of the processor's SpDeque is not empty), the obtained node becomes the processor's new assigned node. However, if the public part of the processor's SpDeque is also empty, the processor becomes a thief and begins a stealing phase. During stealing phases, thieves target victims uniformly at random and attempt to steal work from the top of their SpDeques. To keep the private part of SpDeques entirely private, steal attempts are only allowed to access the public part. Thus, when a thief attempts to steal work from a victim's SpDeque whose public part is empty (illustrated in Figure 1a), the steal attempt simply fails and the thief does not obtain work. In such case, the thief updates a victim's flag (referred to as the *targeted* flag) to (asynchronously) notify the victim that the public part of its SpDeque is empty (more on this ahead). When the owner of the SpDeque realizes it was notified (by checking the value of its *targeted* flag), it tries to transfer a node from the private part of its SpDeque to the public part. If the private part is not empty then a node is transferred, in which case we say that the transferred node became *stealable* (illustrated in Figure 1b).

## 2.1 Specification of the Lock-Free Split Deque

We now present the specification of an SpDeque object, along with its associated relaxed semantics. Being the behavior of SpDeques similar to the behavior of concurrent deques, the SpDeque's relaxed semantics are comparable to the relaxed deque semantics presented in [8].

An SpDeque object meeting the relaxed semantics supports five methods: 1. *push* — Pushes a node into the bottom of the SpDeque's private part. 2. *pop* — Removes and returns a node from the bottom of the SpDeque's private part, if that part is not empty. Otherwise, returns the special value RACE. 3. *updateBottom* — Transfers the topmost node from the private part of the SpDeque into the bottom of the public part, and does not return a value. The invocation has no effect if the private part of the SpDeque is empty. 4. *popBottom* — Removes and returns the bottom-most node from the public part of the SpDeque. If the SpDeque is empty, the invocation has no effect and EMPTY is returned. 5. *popTop* — Attempts to remove and return the topmost node from the public part of the SpDeque. If the public part is empty, the invocation has no effect and the value EMPTY is returned. If the invocation aborts, it has no effect and the value ABORT is returned.

An SpDeque implementation is constant-time iff any invocation to each of these methods takes at most a constant number of steps to return. Say that a set of invocations to an SpDeque's methods meets the relaxed semantics iff there is a set of *linearization times* for the corresponding non-aborting invocations such that: 1. Every non-aborting invocation's linearization time lies within the beginning and completion times of the respective invocation; 2. No linearization times associated with distinct non-aborting invocations coincide; 3. The return values for the non-aborting invocations are consistent with a serial execution of the methods in the order given by the linearization times of the corresponding non-aborting invocations; and 4. For each aborted *popTop* invocation $x$ to an SpDeque $d$, there exists another invocation removing the topmost item from $d$ whose linearization time falls between the beginning and completion times of invocation $x$.

## 2.2 The Low Cost Work Stealing Algorithm

Algorithm 1 depicts the specification of the LCWS algorithm. Each processor owns an SpDeque that uses to store its attached nodes and, additionally, owns a *targeted* flag that stores a Boolean value. This flag is used to implement an asynchronous notification mechanism that allows thieves to request their victims to expose work, allowing it to be stolen. Note that, since the main focus of this paper is on obtaining upper bounds on the synchronization costs for LCWS, we have to explicit all possible sources of synchronization.

Although the *targeted* flag of each processor can be simultaneously accessed by multiple processors, to ensure the algorithm's correctness it suffices to guarantee that no read nor write operation to a processor's *targeted* flag is cached.

Before a computation's execution begins, every processor sets its *assigned* node to NONE and its *targeted* flag to FALSE. To start the execution, one of the processors gets the root node assigned.

As we will see, the behavior of LCWS is similar to the original Work Stealing algorithm. Consider

**Algorithm 1** The CWS algorithm.

```
1: procedure SCHEDULER
2:   while computation not terminated do
3:     if self.targeted then
4:       self.spdeque.updateBottom()
5:       self.targeted ← FALSE
6:     end if
7:     if ValidNode(assigned) then
8:       enabled ← execute(assigned)
9:       if length(enabled) $> 0$ then
10:        assigned ← enabled[0]
11:        if length(enabled) $= 2$ then
12:          self.spdeque.push(enabled[1])
13:        end if
14:      else
15:        assigned ← self.spdeque.pop()
16:        if assigned = RACE then
17:          assigned ← self.spdeque.popBottom()
18:        end if
19:      end if
20:    else
21:      self.WorkMigration()
22:    end if
23:  end while
24: end procedure

25: procedure WORKMIGRATION
26:   victim ← UniformlyRandomProcessor()
27:   assigned ← victim.spdeque.popTop()
28:   if assigned = EMPTY then
29:     victim.targeted ← TRUE
30:   end if
31: end procedure

32: function VALIDNODE(node)
33:   return node ≠ EMPTY and node ≠ ABORT and node ≠ NONE
34: end function
```

some processor $p$ working on a computation scheduled by LCWS, and some iteration of the scheduling loop that $p$ executes (corresponding to lines 2 to 23 of Algorithm 1). First, $p$ reads the value of its *targeted* flag to check if it has been notified by some thief. If $p$'s *targeted* flag is set to TRUE (*i.e.* if $p$ was notified), the processor tries to make a node stealable, by invoking *updateBottom* to its SpDeque. After that, and regardless of that invocation's outcome, $p$ resets its *targeted* flag back to FALSE. The subsequent behavior of $p$ depends on whether it has an assigned node.

- If $p$ has an assigned node, $p$ executes the node. From this execution, either zero, one or two nodes can be enabled.

  **Zero nodes enabled** The processor tries to fetch the bottommost node stored in its SpDeque. To that end, $p$ first tries to fetch a node from the bottom of its SpDequeś private part (line 15). If $p$ finds that part empty, it then tries to fetch a node from the public part (line 17). If this part is also empty, $p$ becomes a thief and starts a work stealing phase. On the other hand, if $p$ successfully fetched a node from any of the parts of its SpDeque, then the returned node becomes $p$'s new assigned node.

  **One node enabled** The enabled node becomes $p$'s new assigned node (line 10).

  **Two nodes enabled** One of the enabled nodes becomes $p$'s new assigned node, whilst the other is pushed into the bottom of the private part of $p$'s SpDeque (line 12).

- If $p$ does not have an assigned node, it is searching for work. In this situation, the processor first targets, uniformly at random, a victim processor and then attempts to steal work from the public part of the victim's SpDeque (lines 26 and 27). If the attempt is successful, the stolen node becomes $p$'s new assigned node. If the attempt aborts, $p$ simply gives up on the steal attempt. For last, if $p$ finds the public part of the victim's SpDeque empty it sets the victim's *targeted* flag to TRUE (line 29), notifying the victim that it found the public part of the victim's SpDeque empty.

## 2.3 A Split Deque Implementation

Algorithm 2 depicts a possible implementation of a Lock Free SpDeque, based on the concurrent deque implementation given in [8]. Each SpDeque object has four instance variables: 1. *entries*: an array of ready nodes, 2. *privateBottom*: the index below the bottommost node of the SpDeque, 3. *officialBottom*: the index below the bottommost node of the SpDeque's public part, and 4. *age*:

**Algorithm 2** The SpDeque implementation

```
    $privateBottom \leftarrow 0$  // private field
    $entries \leftarrow \{\}$  // private read-write, public read-only
    $officialBottom \leftarrow 0$  // private read-write, public read-only
    $age \leftarrow \{0, 0\}$  // public field

 1: procedure PUSH($node$)
 2:   $pBot \leftarrow self.privateBottom$
 3:   $self.entries[pBot] \leftarrow node$
 4:   $self.privateBottom \leftarrow pBot + 1$
 5: end procedure

 6: procedure POP
 7:   $pBot \leftarrow self.privateBottom$
 8:   if $pBot = self.officialBottom$ then
 9:     return RACE
10:   end if
11:   $pBot \leftarrow pBot - 1$
12:   $node \leftarrow self.entries[pBot]$
13:   $self.privateBottom \leftarrow pBot$
14:   return $node$
15: end procedure

16: procedure POPTOP
17:   $oldAge \leftarrow self.age$
18:   $oldBottom \leftarrow self.officialBottom$
19:   if $oldBottom \leq oldAge.top$ then
20:     return EMPTY
21:   end if
22:   $node \leftarrow self.entries[oldAge.top]$
23:   $newAge \leftarrow oldAge$
24:   $newAge.top \leftarrow newAge.top + 1$
25:   if CAS($age$, $oldAge$, $newAge$) = SUCCESS then
26:     return $node$
27:   end if
28:   return ABORT
29: end procedure

30: procedure UPDATEBOTTOM
31:   $pBot \leftarrow self.privateBottom$
32:   $oBot \leftarrow self.officialBottom$
33:   if $pBot > oBot$ then
34:     $oBot \leftarrow oBot + 1$
35:   end if
36:   $self.officialBottom \leftarrow oBot$
37: end procedure

38: procedure POPBOTTOM
39:   $oBot \leftarrow self.officialBottom$
40:   if $oBot = 0$ then
41:     return EMPTY
42:   end if
43:   $oBot \leftarrow oBot - 1$
44:   $self.officialBottom \leftarrow oBot$
45:   $node \leftarrow self.entries[oBot]$
46:   $oldAge \leftarrow age$
47:   if $oBot > oldAge.top$ then
48:     return $node$
49:   end if
50:   $self.officialBottom \leftarrow 0$
51:   $self.privateBottom \leftarrow 0$
52:   $newAge.top \leftarrow 0$
53:   $newAge.tag \leftarrow oldAge.tag + 1$
54:   if $oBot = oldAge.top$ then
55:     if CAS($age$, $oldAge$, $newAge$) = SUCCESS then
56:       return $node$
57:     end if
58:   end if
59:   $self.age \leftarrow newAge$
60:   return EMPTY
61: end procedure
```

composed of two fields: *top*, which corresponds to the index right below the topmost node of the SpDeque's public part; and *tag*, which is only used to ensure correction. Upon creation, the instance variables *privateBottom*, *officialBottom*, and both fields of the instance variable *age* are set to 0.

We say that a set of invocations is *good* if and only if the methods *push*, *pop*, *updateBottom* and *popBottom* are never invoked concurrently. For LCWS, as only the owner of each SpDeque can invoke these methods, it is easy to deduce that all sets of invocations issued by the algorithm are good. Furthermore, we claim that the implementation depicted in Algorithm 2 is constant-time and meets the relaxed semantics (defined in Section 2.1) on any good set of invocations. However, even though all methods are composed by a small number of instructions and none includes a loop, proving this claim is not a straightforward task because all possible execution interleaves have to be considered. Moreover, as the main focus of this study is not related with programs' verification, the proof of this claim falls out of the scope of this paper. Yet, we remark that the proposed implementation is a trivial extension of the deque implementation presented in [8], which has been proven in [14] to be a correct implementation, meeting the relaxed deque semantics on any set of invocations made by the Work Stealing algorithm. For this reason, throughout this paper we assume that for any set of invocations issued by the LCWS algorithm, the relaxed semantics is always satisfied.

**Lemma 2.1.** *No invocation to push requires a MFence instruction.*

*Proof.* Since the *push* method operates only once over a single publicly accessible field (*entries*) of the SpDeque's state, no MFence instructions are required. ■

**Lemma 2.2.** *No invocation to pop requires a MFence instruction.*

*Proof.* Any invocation to the *pop* method only reads from two publicly accessible fields of the SpDeque's state, namely $officialBottom$ (line 8) and *entries* (line 12). However, due to a data dependency, no re-ordering between these read operations may occur, and so, no MFence instructions are required. ■

The dag of a computation is dynamically unfolded during its execution. If the execution of a node $u$ *enables* another node $u'$, then $(u, u')$ is an *enabling edge* and refer to node $u$ as the *designated parent* of $u'$. Refer to the tree formed by the enabling edges of a particular execution of a dag by *enabling tree*, and denote the depth of a node $u$ within this tree by $d(u)$. Define the weight of $u$ as $w(u) = T_\infty - d(u)$. Similarly to [8], the analysis is made in an *a posteriori* fashion, allowing us to refer to the enabling tree generated by a computation's execution.

The following corollary is a direct consequence of the standard properties of deques (a full proof can be found in the appendix (Lemma 6.1)).

**Corollary 2.3.** *Let $v_1, \ldots, v_k$ denote the nodes stored in some processor $p$'s SpDeque, ordered from the bottom to the top, at some moment during the execution of LCWS. Moreover, let $v_0$ denote $p$'s assigned node (if any). Then, we have $w(v_0) \leq w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

## 3 Analysis

In this section we obtain bounds on the expected execution time of computations using LCWS, and on the total expected synchronization costs incurred by the scheduler. The analysis we make follows the same overall idea as the one given in [8]. Due to space restrictions, the proofs of our claims were placed in the appendix.

Define a *scheduling iteration* as a sequence of instructions executed by a processor corresponding to a particular iteration of the scheduling loop (lines 2 to 23 of Algorithm 1). As in [8], we introduce the concept of *milestone*: an instruction within the sequence executed by a processor is a milestone iff it corresponds to a node's execution (line 8) or to the return of a call to *WorkMigration* (line 31). Taking into account the definition of a scheduling iteration it is clear that any scheduling iteration of the algorithm includes a milestone. Refer to iterations whose milestone corresponds to a node's execution as *busy iterations*, and refer to the remainder as *idle iterations*. As one might note, if a processor has an assigned node at the beginning of an iteration's execution, the iteration is a busy one, and, otherwise, the iteration is an idle one. By observing the scheduling loop (lines 2 to 23 of Algorithm 1), and taking into account that the SpDeque's implementation is constant time, it is clear that any scheduling iteration is composed of a constant number of instructions. It then follows that any processor executes at most a constant number of instructions between two consecutive milestones. Throughout the analysis, let $C$ denote a constant that is large enough to guarantee that any sequence of instructions executed by a processor with length at least $C$ includes a milestone.

We can now bound the execution time of a computation depending on the number of idle iterations that take place during that computation's execution. The proof of the following result can be found in the appendix (Lemma 6.3).

**Lemma 3.1.** *Consider any computation with work $T_1$ executed by $P$ processors using LCWS. The execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where $I$ denotes the number of idle iterations executed by processors.*

Now, we prove that the total synchronization used by LCWS only depends on the number of idle iterations (proofs in appendix (Lemmas 6.4 and 6.5, respectively)).

**Lemma 3.2.** *Consider a processor $p$ executing a busy iteration, such that $p$'s targeted flag is set to* FALSE *when $p$ checks it at the beginning of the iteration. If the execution of $p$'s assigned node enables one or more nodes, or, if the private part of $p$'s SpDeque is not empty, then, no MFence instruction is required during the execution of the iteration.*

**Lemma 3.3.** *Consider any computation executed by the LCWS algorithm, using $P$ processors. The total number of CAS instructions executed by processors that were issued by the LCWS algorithm during a computation's execution is at most $O(I + P)$, where $I$ denotes the total number of idle iterations executed by processors. Moreover, the total number of MFence instructions executed by LCWS is at most $O(I + P)$.*

The rest of the analysis focuses on bounding the number of idle iterations that take place during a computation's execution, and follows the same general arguments as the analysis presented in [8].

We say that a node $u$ is *stealable* if $u$ is stored in the public part of some processor's SpDeque. Furthermore, we denote the set of ready nodes at some step $i$ by $R_i$. Consider any node $u \in R_i$. The potential associated with $u$ at step $i$ is denoted by $\phi_i(u)$ and is defined as

$$\phi_i(u) = \begin{cases} 4^{3w(u)-2} & \text{if } u \text{ is assigned} \\ 4^{3w(u)-1} & \text{if } u \text{ is stealable} \\ 4^{3w(u)} & \text{otherwise} \end{cases}$$

The total potential at step $i$, denoted by $\Phi_i$, corresponds to the sum of potentials of all the nodes that are ready at that step: $\Phi_i = \sum_{u \in R_i} \phi_i(u)$. The next result is a formalization of the arguments already given in [8], but considering our potential function (proof in appendix (Lemma 6.6)).

**Lemma 3.4.** *Consider some node $u$, ready at step $i$ during the execution of a computation.*
1. *If $u$ gets assigned to a processor at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$.*
2. *If $u$ becomes stealable at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$.*
3. *If $u$ was already assigned to a processor and gets executed at that step $i$, the potential drops by at least $\frac{47}{64}\phi_i(u)$.*

For the remainder of the analysis, we make use of a few more definitions, first introduced in [8]. We denote the set of ready nodes attached to some processor $p$ (*i.e.* the ready nodes in $p$'s SpDeque together with the node it has assigned, if any) at the beginning of some step $i$ by $R_i(p)$. Furthermore, we define the total potential associated with $p$ at step $i$ as the sum of the potentials of each of the nodes that is attached to $p$ at the beginning of that step: $\Phi_i(p) = \sum_{u \in R_i(p)} \phi_i(u)$.

For each step $i$, we partition the processors into two sets, $D_i$ and $A_i$, where the first is the set of all processors whose SpDeque is not empty at the beginning of step $i$ and the second is the set of all other processors. Thus, the potential at any step $i$, denoted by $\Phi_i$, is composed by the potential associated with each of these two partitions: $\Phi_i = \Phi_i(D_i) + \Phi_i(A_i)$, where $\Phi_i(D_i) = \sum_{p \in D_i} \Phi_i(p)$ and $\Phi_i(A_i) = \sum_{p \in A_i} \Phi_i(p)$.

The following lemmas are a consequence of Corollary 2.3 and the potential function's properties (proofs in appendix: Lemmas 6.7 and 6.8).

**Lemma 3.5.** *Consider any step $i$ and any processor $p \in D_i$. The topmost node $u$ in $p$'s SpDeque contributes at least $\frac{4}{5}$ of the potential associated with $p$. That is, $\phi_i(u) \geq \frac{4}{5}\Phi_i(p)$.*

**Lemma 3.6.** *Suppose a thief processor $p$ chooses a processor $q \in D_i$ as its victim at some step $j$, such that $j \geq i$ (i.e. a steal attempt of $p$ targeting $q$ occurs at step $j$). Then, at step $j + 2C$, the potential decreased by at least $\frac{3}{5}\Phi_i(q)$ due to either the assignment of the topmost node in $q$'s SpDeque, or for making the topmost node of $q$'s SpDeque become stealable.*

The next result follows from Lemma 3.6 and means that, with a constant probability, for each $P$ idle iterations the total potential associated with the processors whose SpDeque is not empty drops by a constant factor (proof in appendix (Lemma 6.10)).

**Lemma 3.7.** *Consider any step $i$ and any later step $j$ such that at least $P$ idle iterations occur from $i$ (inclusive) to $j$ (exclusive). Then, we have $P\left\{\Phi_i - \Phi_{j+2C} \geq \frac{3}{10}\Phi_i\left(D_i\right)\right\} > \frac{1}{4}$.*

With this, we can finally bound the expected number of idle iterations. The result follows from Lemma 3.7 (proof in appendix (Lemma 6.11)).

**Lemma 3.8.** *Consider any computation with work $T_1$ and span $T_\infty$ executed by LCWS using $P$ processors. The expected number of idle iterations is at most $O\left(PT_\infty\right)$. Moreover, with probability at least $1-\varepsilon$ the number of idle iterations is at most $O\left(\left(T_\infty + \ln\left(\frac{1}{\varepsilon}\right)\right)P\right)$.*

Finally, we obtain bounds on the expected execution time of the LCWS algorithm.

**Theorem 3.9.** *Consider any computation with work $T_1$ and span $T_\infty$ executed by the LCWS algorithm with $P$ processors. The expected execution time is at most $O\left(\frac{T_1}{P} + T_\infty\right)$. Furthermore, with probability at least $1-\varepsilon$ the execution time is at most $O\left(\frac{T_1}{P} + T_\infty + \ln\left(\frac{1}{\varepsilon}\right)\right)$.*

*Proof.* This result follows from Lemma 3.1 and Lemma 3.8. ■

**Theorem 3.10.** *Consider any computation with work $T_1$ and span $T_\infty$ executed by the LCWS algorithm with $P$ processors, and let $C_{CAS}$ and $C_{MFence}$ respectively denote the synchronization costs incurred by the execution of a CAS and MFence instructions. Then, the expected total synchronization cost of LCWS is at most $O\left(\left(C_{CAS} + C_{MFence}\right)PT_\infty\right)$. Moreover, with probability at least $1-\varepsilon$ the total synchronization cost of LCWS is at most $O\left(\left(C_{CAS} + C_{MFence}\right)P\left(T_\infty + \ln\left(\frac{1}{\varepsilon}\right)\right)\right)$.*

*Proof.* Lemma 3.3 and Lemma 3.8 prove this theorem. ■

## 4 Related work

Prominent research has taken considerable steps towards avoiding the synchronization overheads, from the implementation of concurrent deques, that heavily impact execution performance. Michael *et al.* propose an idempotent version of Work Stealing that eliminates the overheads of synchronization by relaxing the semantics of work queues [27]. Concretely, rather than using the conventional *exactly-once* semantics, the authors present several concurrent data structures only satisfying *at-least-once* semantics. By using these structures, processors no longer have to incur in expensive synchronization overheads when operating locally, which, as the authors show, can be extremely beneficial. Unfortunately this strategy suffers from two limitations: on one hand, for non-idempotent computations the synchronization overheads are moved to the computation itself, as it is necessary to ensure correctness; on the other, for idempotent computations these semantics can lead to situations where some computationally heavy tasks are executed more than once, limiting the scheduler's performance.

Hiraishi *et al.* suggested that processors should behave as in a sequential execution, thus eliminating all overheads inherent from parallelism [22]. Under their scheme, deques are kept entirely private and processors only permit parallelism when an idle processor requests work. Upon such request, the busy processor backtracks to the last point where it could have spawned a task, spawns the task, offers it to the requesting processor, and, finally, proceeds with the execution. Since work requests are usually rare, the gains of eliminating synchronization for local operation greatly surpasses the extra overheads caused by using entirely private deques, which for this algorithm come from backtracking. This remark is key and motivated the efforts for reducing synchronization overheads to focus on the elimination of synchronization for local operation.

Various proposals have been made with the goal of eliminating synchronization for local deque accesses, by making deques partly or even entirely private [2,26,28,31,32]. Acar *et al.* presented and studied two Work Stealing algorithms — *sender-* and *receiver-initiated* — that avoid synchronization by making deques entirely private to each processor [2]. In addition to promising empirical results, their theoretical analysis showed that the expected execution time for both algorithms can be somewhat competitive with Work Stealing algorithms that use concurrent deques. For the sender-initiated algorithm, busy processors now have to periodically search for idle ones, and thus synchronization is replaced by communication. Regarding the receiver-initiated algorithm the trade-off is more subtle: since deques are kept entirely private to each processor, when an idle processor requests work from a busy one, it has to perform a handshake with the busy processor. For multi-programmed environments these handshakes can be harmful if, for example, the busy processor — in this case, the busy process — is not currently scheduled by the Operating System. Additionally, for this approach, most load balancing burdens are transferred to the busy processors, who, to handle a work request, have to halt their work, pop a thread from the top of their deque and send it to the requesting processor.

The first use of split queues for avoiding unnecessary synchronization was, to the best of our knowledge, by Endo *et al.* in [19]. In this study, the authors presented an implementation of a scalable garbage collector system that, by using clever load balancing techniques and split queues to avoid unnecessary synchronization overheads, achieves high performances even for large scale machines. Later, and as already mentioned, Dinan *et al.* studied Work Stealing under a distributed environment and proposed the use of SpDeques to avoid synchronization for local deque accesses [17,18]. More recently, Dijk *et al.* studied the effectiveness of SpDeques on shared memory environments [32,33]. We remark that our algorithm has some similarities with Dijk *et al.*'s, since thieves also use flags to inform busy processors that they should add work to the public part of their deques, allowing it to be stolen. However, in their algorithm, and in contrast with our approach, busy processors only check for requests each time they access their deque. As a consequence, the scheduler is not appropriate for generic computations because the frequency at which busy processors may permit load balancing depends on the structure of the computation. In particular, for computations whose workload is mostly composed by large sequential threads, processors will rarely check for the necessity of load balancing, giving a very small room for parallelism. Finally, when a busy processor realizes it was targeted by a steal attempt, it makes at least half of its work to become public. Even though this strategy is useful to mitigate the limitation we noted above, it severely increases the synchronization costs incurred by the algorithm. For instance, busy processors, by making half of their work to become public, have to incur extra synchronization overheads for each time they have to fetch work from the non-private part of their deques.

Morrison *et al.* studied alternative designs to the synchronization protocols used by Work Stealing schedulers, considering the architectures of modern *TSO* processors [28]. The authors found that, by taking into account the actual implementation of microprocessors, MFence instructions can be fully eliminated while maintaining correctness. In their algorithm, thieves can only steal work from a victim if such work is stored at a safe distance, far enough from the bottom of the victim's deque to avoid any data race. This safe distance is computed *a priori*, taking into account the size of the microprocessor's internal store buffer to determine the minimum safe distance to avoid any possible conflicts. With this strategy, not only thieves can asynchronously take work from their victims, but processors can also access their deques locally without requiring any synchronization. Unfortunately, their scheme suffers from a limitation similar to Dijk *et al.*'s, as the bottommost threads within a processor's deque cannot to be stolen, meaning their strategy is not appropriate for generic computations. Nevertheless, the authors' showed that this technique (similar to using SpDeques) substantially outperforms Work Stealing algorithms that use concurrent deques.

Lifflander *et al.* studied the execution of iterative over-decomposed applications [26] and proposed, among others, a message-based retentive Work Stealing algorithm adapted for the execution of iterative workloads on large scale distributed settings. To avoid synchronization overheads and improve the overall performance of the scheduler, their Work Stealing algorithm uses SpDeques. Their evaluation showed that Work Stealing (with SpDeques) is a practical algorithm even for systems with as much as 163.940 processors, for which their Work Stealing algorithm achieved a remarkable result of more than 91% of efficiency. A distinction between the Work Stealing algorithm proposed by Lifflander *et al.* and LCWS is that our algorithm keeps processors' deques entirely private by default. As we will discuss in Section 5, this fact is key for guaranteeing that a 1-Processor execution of our algorithm requires no synchronization.

Tzannes *et al.* proposed a scheduling algorithm where each processor keeps all of its work entirely private, except for the topmost node that is kept stored in a shared cell [31]. However, since the algorithm always ensures that the topmost node is shared, it does not behave appropriately for various types of computations in which processors access the topmost nodes of their deques often. As mentioned in [2], a similar limitation has been identified for the Chase-Lev Deque [16].

## 5 Conclusion

In this paper we studied a Work Stealing algorithm that uses SpDeques to reduce synchronization overheads. We showed that the expected execution time of our algorithm is asymptotically optimal, and that the expected total synchronization of the algorithm is $O\left(PT_\infty\left(C_{CAS}+C_{MFence}\right)\right)$. To (informally) justify the tightness of our bounds, we recall that, for LCWS, the expected number of (successful and unsuccessful) steal attempts is $O\left(PT_\infty\right)$. Thus, by noting that the public part of an SpDeque is essentially a concurrent deque, and, by taking into account the impossibility of eliminating all synchronization from the implementation of concurrent deques while maintaining their correction (see [9]), we conclude that the synchronization bounds we have obtained for LCWS are tight.

In addition to an asymptotically optimal expected execution time, we claim that the expected total communication of LCWS is the same as for the Work Stealing algorithm given in [11], for which, in turn, the expected total communication is known to be existentially optimal [11, 34]. Despite being trivial to prove this claim after having obtained bounds on the expected number of steal attempts, such proof falls out of the scope of this paper as it requires the introduction of various extra definitions (such as the notion of threads) and additional assumptions on computations' structure. Nevertheless, we note that by using the exact same reasoning made in the proof of [11, Theorem 14] we could obtain the proof of our claim, meaning that the expected total communication of LCWS is also existentially optimal.

To conclude, note that, by taking into account our assumptions regarding computations' structure, we can create computations for which $T_1 = O\left(2^{T_\infty}\right)$. Since for such computations the number of deque accesses grows linearly with the total amount of work, $T_1$, our result means that the use of SpDeques allows to reduce by an almost exponential factor the synchronization present in conventional Work Stealing algorithms.

## References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 219–228, 2013.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Pasl: Parallel algorithm scheduling library, 2016. [Online; accessed 21-January-2016].

[4] Kunal Agrawal, Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–7, 2007.

[5] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen-Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3), 2008.

[6] Noga Alon and Joel Spencer. *The Probabilistic Method.* John Wiley, 1992.

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, 1998.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[9] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498, 2011.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[11] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[12] Robert D Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 266–267. ACM, 1998.

[13] Robert D Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, Citeseer, 1999.

[14] Robert D Blumofe, C Greg Plaxton, and Sandip Ray. Verification of a concurrent deque implementation. *University of Texas at Austin, Austin, TX*, 1999.

[15] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538. ACM, 2005.

[16] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 21–28, 2005.

[17] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 586–593, 2008.

[18] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009.

[19] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1997, November 15-21, 1997, San Jose, CA, USA*, page 48, 1997.

[20] Karl-Filip Faxén. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2009.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223. ACM, 1998.

[22] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 55–64, 2009.

[23] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[24] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[25] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 522–527, 2009.

[26] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kalé. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *The 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands - June 18 - 22, 2012*, pages 137–148, 2012.

[27] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 45–54, 2009.

[28] Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 413–426, 2014.

[29] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 71–82, 2016.

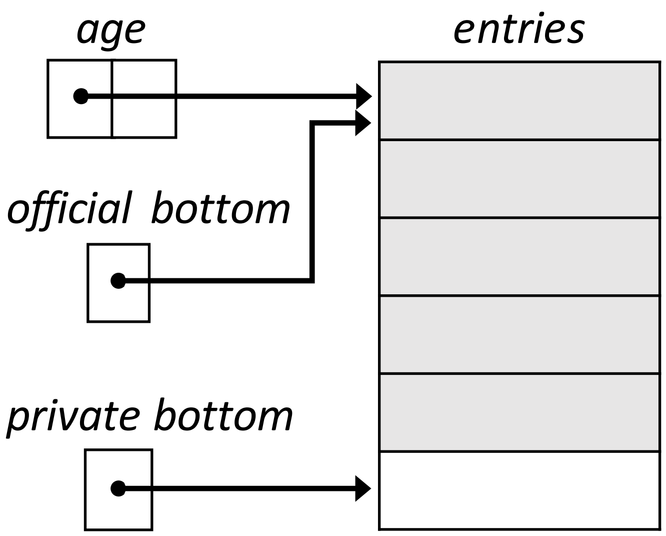
[30] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, pages 291–302, 2010.

[31] Alexandros Tzannes. Enhancing productivity and performance portability of general-purpose parallel programming. 2012.

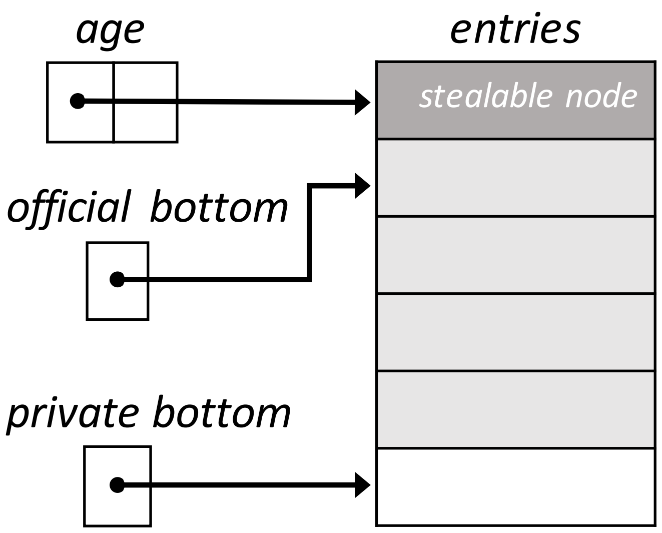
[32] Tom van Dijk and Jaco C. van de Pol. Lace: Non-blocking split deque for work-stealing. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 206–217, 2014.

[33] Thijs van Ede. Certainty in lockless concurrent algorithms: an informal proof of lace. 2015.

[34] I-Chen Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 151–162, 1991.



(a) An SpDeque with no stealable nodes



(b) An SpDeque with one stealable node

Figure 1: The Split Deque

## 6 Appendix: Proofs

Due to space constraints, most of the proofs of our claims are placed in this appendix. Nevertheless, if this paper is accepted, we will publish a full version of this paper, with proofs, in a freely accessible on-line repository.

The following lemma is key for the analysis of LCWS. An analogous result has already been proved for concurrent deques (see [8, Lemma 3]). For the sake of completion we present its proof, which is a simple transcription of original proof of [8, Lemma 3], adapted for SpDeques.

**Lemma 6.1** (Structural Lemma for SpDeques)**.** *Let $v_1, \ldots, v_k$ denote the nodes stored in some processor p's SpDeque, ordered from the bottom of the SpDeque to the top, at some point in the linearized execution of LCWS. Moreover, let $v_0$ denote p's assigned node (if any), and for $i = 0, \ldots, k$ let $u_i$ denote the designated parent of $v_i$ in the enabling tree. Then, for $i = 1, \ldots, k$, $u_i$ is an ancestor of $u_{i-1}$ in the enabling tree, and despite $v_0$ and $v_1$ may have the same designated parent (i.e. $u_0 = u_1$), for $i = 2, 3, \ldots, k$, $u_{i-1} \neq u_i$, i.e. that the ancestor relationship is proper.*

*Proof.* Fix a particular SpDeque. The SpDeque state and assigned node only change when the assigned node is executed or a thief performs a successful steal. We prove the claim by induction on the number of assigned-node executions and steals since the SpDeque was last empty. In the base case, if the SpDeque is empty, then the claim holds vacuously. We now assume that the claim holds before a given assigned-node execution or successful steal, and we will show that it holds after. Specifically, before the assigned-node execution or successful steal, let $v_0$ denote the assigned node; let $k$ denote the number of nodes in the SpDeque; let $v_1, \ldots, v_k$ denote the nodes in the SpDeque ordered from the bottom to top; and for $i = 0, \ldots, k$, let $u_i$ denote the designated parent of $v_i$. We assume that either $k = 0$, or for $i = 1, \ldots, k$, node $u_i$ is an ancestor of $u_{i-1}$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. After the assigned-node execution or successful steal, let $v_0{}'$ denote the assigned node; let $k'$ denote the number of nodes in the SpDeque; let $v_1{}', \ldots, v_k{}'$ denote the nodes in the SpDeque ordered from bottom to top; and for $i = 1, \ldots, k'$, let $u_i{}'$ denote the designated parent of $v_i{}'$. We now show that either $k' = 0$, or for $i = 1, \ldots, k'$, node $u_i{}'$ is an ancestor of $u_{i-1}{}'$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$.

Consider the execution of the assigned node $v_0$ by the owner.

If the execution of $v_0$ enables 0 children, then the owner pops the bottommost node off its SpDeque and makes that node its new assigned node. If $k = 0$, then the SpDeque is empty; the owner does not get a new assigned node; and $k' = 0$. If $k > 0$, then the bottommost node $v_1$ is popped and becomes the new assigned node, and $k' = k - 1$. If $k = 1$, then $k' = 0$. Otherwise, $k' = k - 1$. We now rename the nodes as follows. For $i = 0, \ldots, k'$, we set $v_i{}' = v_{i+1}$ and $u_i{}' = u_{i+1}$. We now observe that for $i = 1, \ldots, k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree.

If the execution of $v_0$ enables 1 child $x$, then $x$ becomes the new assigned node; the designated parent of $x$ is $v_0$; and $k' = k$. If $k = 0$, then $k' = 0$. Otherwise, we can rename the nodes as follows. We set $v_0{}' = x$; we set $u_0{}' = v_0$; and for $i = 1, \ldots, k'$, we set $v_i{}' = v_i$ and $u_i{}' = u_i$. We now observe that for $i = 1, \ldots, k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_1{}'$ is a proper ancestor of $u_0{}'$ in the enabling tree follows from the fact that $(u_0, v_0)$ is an enabling edge.

In the most interesting case, the execution of the assigned node $v_0$ enables 2 children $x$ and $y$, with $x$ being pushed onto the bottom of the SpDeque and $y$ becoming the new assigned node. In this case, $(v_0, x)$ and $(v_0, y)$ are both enabling edges, and $k' = k + 1$. We now rename the nodes as follows. We set $v_0{}' = y$; we set $u_0{}' = v_0$; we set $v_1{}' = x$; we set $u_1{}' = v_0$; and for $i = 2, \ldots, k'$, we set $v_i{}' = v_{i-1}$ and $u_i{}' = u_{i-1}$. We now observe that $u_1{}' = u_0{}'$, and for $i = 2, \ldots, k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_2{}'$ is a proper ancestor of $u_1{}'$ in the enabling tree follows from the fact that $(u_0, v_0)$ is an enabling edge.

Finally, we consider a successful steal by a thief. In this case, the thief pops the topmost node $v_k$ off the SpDeque, so $k' = k - 1$. If $k = 1$, then $k' = 0$. Otherwise, we can rename the nodes as follows. For $i = 0, \ldots, k'$, we set $v_i{}' = v_i$ and $u_i{}' = u_i$. We now observe that for $i = 1, \ldots, k'$, node $u_i{}'$ is an ancestor of $u_{i-1}{}'$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. ■

**Corollary 6.2.** *If $v_0$, $v_1$, …, $v_k$ are as defined in the statement of Lemma 6.1, then we have $w(v_0) \leq w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

Now, we obtain a bound on the execution time of a computation depending on the number of idle iterations that take place during that computation's execution. The following result is a trivial variant of [8, Lemma 5] but considering the LCWS algorithm.

**Lemma 6.3.** *Consider any computation with work $T_1$ executed by $P$ processors, under LCWS. The execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where $I$ denotes the number of idle iterations executed by processors.*

*Proof.* Consider two buckets to which we add tokens during the computation's execution: the *busy* bucket and the *idle* bucket. At the end of each iteration, every processor places a token into one of these buckets. If a processor executed a node during the iteration, it places a token into the busy bucket, and otherwise, it places a token into the idle bucket. Since we have $P$ processors, for each $C$ consecutive steps, at least $P$ tokens are placed into the buckets.

Because, by definition, the computation has $T_1$ nodes, there will be exactly $T_1$ tokens in the busy bucket when the computation's execution ends. Moreover, as $I$ denotes the number of idle iterations, it also corresponds to the number of tokens in the idle bucket when the computation's execution ends. Thus, exactly $T_1 + I$ tokens are collected during the computation's execution. Taking into account that for each $C$ consecutive steps at least $P$ tokens are placed into the buckets, we conclude the number of steps required to collect all the tokens is at most $C.\left(\frac{T_1}{P} + \frac{I}{P}\right)$. After collecting all the $T_1$ tokens, the computation's execution terminates, implying the execution time is at most $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$. ■

The following result is crucial for the analysis of the synchronization costs of LCWS.

**Lemma 6.4.** *Consider a processor $p$ executing a busy iteration such that $p$'s targeted flag is set to* FALSE *when $p$ checks it at the beginning of the iteration. If the execution of $p$'s assigned node enables one or more nodes, or, if the private part of $p$'s SpDeque is not empty, then, no MFence instruction is required during the execution of the iteration.*

*Proof.* Consider the LCWS algorithm, depicted in Algorithm 1. The first action $p$ takes for the execution of that iteration is checking the value of its *targeted* flag (line 3). Since, by the statement of this lemma $p$'s *targeted* flag is set to FALSE at the moment when $p$ checks the flag's value, $p$ does not enter the *then* branch of the if statement. Moreover, as a consequence of the conditional statement of line 3, there is a control dependency that does not allow the instructions succeeding the conditional expression to be reordered with the evaluation of the condition, implying no memory fence is required until this point.

After that, $p$ checks if it has an assigned node. Again, since the next action $p$ takes depends on its currently assigned node, there is a control dependency from the instruction where $p$ checks if it has a currently assigned node to both branches of the if statement. Thus, no instruction reordering between the evaluation of the condition and any of the instructions succeeding that evaluation can be made, implying no memory fence is required until here.

Because we assumed $p$ was executing a busy iteration, by the definition of a busy iteration $p$ must have an assigned node. Hence, $p$ executes its assigned node. Since the next action $p$ takes depends on the outcome of that node's execution, there is a control dependency between the execution of $p$'s assigned node and the execution of the sequence of instructions corresponding to each of the possible outcomes. Hence, no instruction reordering can be made, implying no memory fence is necessary until this point.

From that node's execution, three outcomes are possible:

**0 nodes enabled** In this case, $p$ invokes the *pop* method to its own SpDeque. By Lemma 2.2, the invocation does not require a memory fence to be issued. Furthermore, the next instruction (line 16) has a data dependency on the value of $p$'s assigned node, for which reason it cannot be reordered with the invocation of the *pop* method and so a memory fence is not required.

Since we have assumed that the private part of $p$'s SpDeque was not empty, it is trivial to conclude that the *pop* invocation returns a node, which immediately becomes $p$'s new assigned node. Thus, after having a new node assigned $p$ takes no further action during the iteration, meaning no memory fence was required for the execution of the iteration in this situation.

**1 node enabled** In this case the enabled node becomes $p$'s new assigned node. Next, $p$ checks the number of nodes that were enabled. The assignment of one of the enabled nodes and the instruction where $p$ checks the number of nodes enabled can be reordered. Fortunately, because *enabled* is a local variable that is solely accessed by $p$, there is no harm for a parallel execution if the instructions are reordered and so no memory fence is required for this case as well. Because $p$ enabled a single node it takes no further action during the iteration, implying the lemma holds in this situation as well.

**2 nodes enabled** Finally, for this situation one of the enabled nodes becomes $p$'s new assigned node. Using the same reasoning as for the case where a single node was enabled, we conclude that no memory fence is required at least until the evaluation of the conditional statement. Because $p$ enabled two nodes, $p$ enters the conditional expression and pushes the other node (*i.e.* the node it did not assign) into the bottom of its SpDeque, by invoking the *push* method. Since there is a control dependency between the execution of this instruction and the evaluation of the condition, no instruction reordering is allowed. Thus, no memory fence is required between these two instructions.

Finally, Lemma 2.1 states that an invocation to the *push* method does not require a memory fence to be issued. Because after the invocation $p$ takes no further action during the iteration, we deduce the lemma holds, concluding its proof.

■

The following lemma is a consequence of Lemma 3.2 and states that the number of memory fences issued during the execution of a computation using LCWS only depends on the number of idle iterations.

**Lemma 6.5.** *Consider any computation executed by the LCWS algorithm, using $P$ processors. The total number of CAS instructions executed by processors that were issued by the LCWS algorithm during a computation's execution is at most $O(I + P)$, where $I$ denotes the number of idle iterations executed by processors. Moreover, the total number of MFence instructions executed by the LCWS algorithm is at most $O(I + P)$.*

*Proof.* By observing Algorithms 1 and 2, it is easy to see that only invocations to *popBottom* or *popTop* methods can lead to the execution of *CAS* instructions. Furthermore, both these methods are invoked at most once per scheduling iteration, and, for both, at most one *CAS* instruction is executed per invocation. Since processors only invoke the *popTop* method when executing idle iterations, the number of *CAS* instructions caused by invocations to *popTop* is $O(I)$. On the other hand, processors only invoke the *popBottom* method during busy iterations where the private part of their SpDeque is empty and the execution of their currently assigned node does not enable any new nodes. Let $p$ denote some processor executing one such iteration. From $p$'s invocation to the *popBottom* method two outcomes are possible:

**A node is returned** In this case the public part of $p$'s SpDeque was not empty implying $p$ had previously transferred a node from the private part of its SpDeque to the public part. By observing Algorithm 1 it is easy to deduce that $p$ only makes these node transfers if some thief had previously set $p$'s *targeted* flag to TRUE. Moreover, because after transferring the node $p$ immediately sets its *targeted* flag back to FALSE, the number of times $p$ makes such node transfers is at most the number of times it is targeted by a steal attempt. Taking into account that processors only make steal attempts during the execution of idle iterations, and make exactly one steal attempt for each such iteration, exactly $I$ steal attempts take place during a computation's execution. As such, the number of *CAS* instructions executed in situations like this one is at most $O(I)$.

**Empty is returned** In this case $p$ will not have an assigned node at the end of the scheduling iteration's execution. Thus, after $p$ finishes executing the iteration two scenarios may occur:

$p$ **executes an idle iteration** For this case we can create a mapping from idle iterations to each busy iteration that precedes an idle iteration, implying there can be at most $O(I)$ such iterations. With this, it is trivial to conclude that the number of *CAS* instructions executed by LCWS for situations equivalent to this one is at most $O(I)$.

**The computation's execution terminates** Because there are exactly $P$ processors, at most $P$ scheduling iterations can precede the end of a computation's execution. Consequently, the number of *CAS* instructions executed for scenarios equivalent to this one is at most $O(P)$.

Summing up all the possible scenarios, the number of *CAS* instructions executed by LCWS is at most $O(I+P)$.

We now turn to the number of memory fences issued during a computation's execution. To that end, we first bound the number of scheduling iterations that can issue memory fences. Consider any scheduling iteration $s$ during a computation's execution, and let $p$ denote the processor that executed the iteration. Iteration $s$ was either an idle or a busy iteration.

$s$ **is an idle iteration** By definition, at most $I$ iterations are idle, implying there are $O(I)$ such iterations that could issue memory fences.

$s$ **is a busy iteration** When $p$ checks its *targeted* flag, one of the two following situations arises:

*targeted* **is true** By observing Algorithm 1 we conclude that such a situation can only occur if another processor $q$ has set $p$'s *targeted* to TRUE, which can only occur if $q$ was executing an idle iteration. After executing the conditional statement, $p$ resets its *targeted* flag back to FALSE. Thus, the total number of busy iterations where a processor has its flag set to *targeted* is at most $I$, because each such iteration can be mapped by an idle iteration. Consequently, the number of iterations similar to this one is at most $O(I)$.

*targeted* **is false** As $p$ is executing a busy iteration, it will execute the node it has assigned. From that node's execution, either 0, 1 or 2 other nodes can be enabled.

- **More than 0 nodes are enabled** Lemma 3.2 implies that no memory fence is issued in this case.
- **0 nodes are enabled** In this case, $p$ cannot immediately assign a new node, because it did not enable any. By Algorithm 1, $p$ will then invoke the *pop* method to its own SpDeque. With this, one of two possible situations arises:
  - **SpDeque's private part is not empty** As a consequence of Lemma 3.2, no memory fence is issued in this case.
  - **SpDeque's private part is empty** In this case, by observing Algorithm 2 we conclude that the invocation returns the special value RACE, implying $p$ will make an invocation to *popBottom* still during that same iteration. From that invocation, two outcomes are possible:
    - **A node is returned** In this situation, $p$ assigns the node. By observing Algorithm 1 it is trivial to conclude that this scenario only arises if some processor previously set $p$'s *targeted* flag to TRUE. As a consequence, $p$ transfered a node from the private part of its SpDeque to the public part. Again, using the same reasoning as for the case where $p$'s *targeted* flag is set to TRUE, we conclude the number of such iterations is at most $O(I)$.
    - **empty is returned** After $p$ finishes executing the current scheduling iteration $s$, two scenarios may occur:
      - $p$ **executes an idle iteration** It is trivial to deduce that we can create a mapping from idle iterations to each iteration satisfying the same conditions as $s$. Thus, there can be at most $O(I)$ such iterations.
      - **The computation's execution terminates** Since there are exactly $P$ processors, at most $P$ scheduling iterations can precede the end of a computation's execution. Consequently, there are at most $P$ scheduling iteration similar to $s$.

Now, we sum up all the scheduling iterations for which memory fences may be required. Accounting with all possible scenarios it follows that in at most $O(I+P)$ scheduling iterations memory fences may be required. Since any scheduling iteration is composed by at most $C$ instructions, at most $C$ memory fences can be required per iteration, implying the number of memory fences required during a computation's execution is at most $O(I+P)$. ■

The following lemma is a formalization of the arguments already given in [8], but considering the potential function we present.

**Lemma 6.6.** *Consider some node $u$, ready at step $i$ during the execution of a computation.*

1. *If $u$ gets assigned to a processor at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$.*
2. *If $u$ becomes stealable at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$.*
3. *If $u$ was already assigned to a processor and gets executed at that step $i$, the potential drops by at least $\frac{47}{64}\phi_i(u)$.*

*Proof.* Regarding the first claim, if $u$ was stealable the potential decreases from $4^{3w(u)-1}$ to $4^{3w(u)-2}$. Otherwise, the potential decreases from $4^{3w(u)}$ to $4^{3w(u)-2}$, which is even more than in the previous case. Given that

$$\begin{aligned} 4^{3w(u)-1} - 4^{3w(u)-2} &= \frac{3}{4} 4^{3w(u)-1} \\ &= \frac{3}{4} \phi_i \left(u\right), \end{aligned}$$

we conclude that if $u$ gets assigned the potential decreases by at least $\frac{3}{4}\phi_i\left(u\right)$.

Regarding the second one, note that $u$ was not stealable (because it became stealable at step $i$). Thus, the potential decreases from $4^{3w(u)}$ to $4^{3w(u)-1}$. It follows,

$$\begin{aligned} 4^{3w(u)} - 4^{3w(u)-1} &= \frac{3}{4} 4^{3w(u)} \\ &= \frac{3}{4} \phi_i \left(u\right) \end{aligned}$$

So, if $u$ becomes stealable, the potential decreases by $\frac{3}{4}\phi_i\left(u\right)$.

We now prove the last claim. Remind that, by our conventions regarding computations' structure, each node within a computation's dag can have an out-degree of at most two. Consequently, each node can be the designated parent of at most two other ones in the enabling tree. Moreover, by definition, the weight of any node is strictly smaller than the weight of its designated parent, since it is deeper in the enabling tree than its designated parent. Consider the three possible scenarios:

**0 nodes enabled** The potential decreased by $\phi_i\left(u\right)$.

**1 node enabled** The enabled node becomes the assigned node of the processor (that executed $u$). Let $x$ denote the enabled node. Since $x$ is the child of $u$ in the enabling tree, it follows

$$\begin{aligned} \phi_i\left(u\right) - \phi_{i+1}\left(x\right) &= 4^{3w(u)-2} - 4^{3w(x)-2} \\ &= 4^{3w(u)-2} - 4^{3(w(u)-1)-2} \\ &= 4^{3w(u)-2}\left(1 - \frac{1}{64}\right) \\ &= \frac{63}{64}\phi_i\left(u\right) \end{aligned}$$

Thus, for this situation, the potential decreases by $\frac{63}{64}\phi_i\left(u\right)$.

**2 nodes enabled** In this case, one of the enabled nodes immediately becomes the assigned node of the processor whist the other is pushed onto the bottom of the SpDeque's private part. Let $x$ denote the enabled node that becomes the processor's new assigned node and $y$ the other enabled node. Since both $x$ and $y$ have $u$ as their designated parent in the enabling tree, we have

$$\begin{aligned} \phi_i\left(u\right) - \phi_{i+1}\left(x\right) - \phi_{i+1}\left(y\right) &= 4^{3w(u)-2} - 4^{3w(x)} - 4^{3w(y)-2} \\ &= 4^{3w(u)-2} - 4^{3((w(u)-1)} - 4^{3(w(u)-1)-2} \\ &= 4^{3w(u)-2}\left(1 - \frac{1}{4} - \frac{1}{64}\right) \\ &= \frac{47}{64}\phi_i\left(u\right) \end{aligned}$$

As such, the potential decreases by $\frac{47}{64}\phi_i\left(u\right)$, concluding the proof of the lemma.

■

The following lemma is a direct consequence of Corollary 2.3 and of the potential function's properties. The result is a variant of [8, Top-Heavy Deques], considering SpDeques instead of deques and our potential function, instead of the original.

**Lemma 6.7.** *Consider any step $i$ and any processor $p \in D_i$. The topmost node $u$ in $p$'s SpDeque contributes at least $\frac{4}{5}$ of the potential associated with $p$. That is, we have*

$$\phi_i(u) \geq \frac{4}{5}\Phi_i(p).$$

*Proof.* This lemma follows from Corollary 2.3. We prove it by induction on the number of nodes within $p$'s SpDeque.

**Base case** As the base case, consider that $p$'s SpDeque contains a single node $u$. The processor itself can either have or not an assigned node. For the second scenario, we have $\phi_i(u) = \Phi_i(p)$. Regarding the first case, let $x$ denote $p$'s assigned node. Corollary 2.3 implies that $w(u) \geq w(x)$. It follows

$$\begin{aligned}\Phi_i(q) &= \phi_i(u) + \phi_i(v)\\ &= 4^{3w(u)-1} + 4^{3w(v)-2}\\ &= 4^{3w(u)-1} + 4^{3w(u)-2}\\ &= 4^{3w(u)-1}\left(1 + \frac{1}{4}\right)\\ &= \frac{5}{4}\phi_i(u)\end{aligned}$$

Thus, if $p$'s SpDeque contains a single node we have $\Phi_i(q) \leq \frac{5}{4}\phi_i(u)$.

**Induction step** Consider that $p$'s SpDeque now contains $n$ nodes, where $n \geq 2$, and let $u$, $x$ denote the topmost and second topmost nodes, respectively, within the SpDeque. For the purpose of induction, let us assume the lemma holds for all the first $n-1$ nodes (*i.e.* without accounting with $u$):

$$\Phi_i(q) - \phi_i(u) \leq \frac{5}{4}\phi_i(x).$$

Corollary 2.3 implies $w(u) > w(x) \equiv w(u) - 1 \geq w(x)$. It follows

$$\begin{aligned}\Phi_i(q) &\leq \frac{5}{4}\phi_i(x) + \phi_i(u)\\ &= \frac{5}{4}4^{3w(x)} + 4^{3w(u)}\\ &\leq \frac{5}{4}4^{3(w(u)-1)} + 4^{3w(u)}\\ &= \frac{5}{4}4^{3w(u)-3} + 4^{3w(u)}\\ &= 4^{3w(u)}\left(1 + \frac{5}{4}4^{-3}\right)\\ &= \frac{261}{256}\phi_i(u)\\ &< \frac{5}{4}\phi_i(u)\end{aligned}$$

Concluding the proof of the lemma.

■

The following result is a consequence of Lemma 3.5.

**Lemma 6.8.** *Suppose a thief processor $p$ chooses a processor $q \in D_i$ as its victim at some step $j$, such that $j \geq i$ (i.e. a steal attempt of $p$ targeting $q$ occurs at step $j$). Then, at step $j + 2C$, the potential decreased by at least $\frac{3}{5}\Phi_i(q)$ due to either the assignment of the topmost node in $q$'s SpDeque, or for making the topmost node of $q$'s SpDeque become stealable.*

*Proof.* Let $u$ denote the topmost node of $q$'s SpDeque at the beginning of step $i$. We first prove that $u$ either gets assigned or becomes stealable.

Three possible scenarios may take place due to $p$'s steal attempt targeting $q$'s SpDeque.

**The invocation returns a node** If $p$ stole $u$, then, $u$ gets assigned to $p$. Otherwise, some other processor removed $u$ before $p$ did, implying $u$ got assigned to that other processor.

**The invocation aborts** Since the SpDeque implementation meets the relaxed semantics on any good set of invocations, and because the LCWS algorithm only makes good sets of invocations, we conclude that some other processor successfully removed a topmost node from $q$'s SpDeque during the aborted steal attempt made by $p$. If the removed node was $u$, $u$ gets assigned to a processor (that may either be $q$, or, some other thief that successfully stole $u$). Otherwise, $u$ must have been previously stolen by a thief or popped by $q$, and thus became assigned to some processor.

**The invocation returns empty** This situation can only occur if either $q$'s SpDeque is completely empty, or if there is no node in the public part of $q$'s SpDeque.

- For the first case, since $q \in D_i$, some processor must have successfully removed $u$ from $q$'s SpDeque. Consequently, $u$ was assigned to a processor.
- If there was no node in the public part of $q$'s SpDeque, $p$ sets $q$'s *targeted* flag to TRUE in a later step $j'$. Recall that, for each $C$ consecutive instructions executed by a processor, at least one corresponds to a milestone. It follows that $j' \leq j + C$. Furthermore, by observing Algorithm 1, we conclude that $q$ will make and complete an invocation to *updateBottom* of its SpDeque in one of the $C$ steps succeeding step $j'$. Thus if $q$'s SpDeque's private part is not empty, a node will become stealable. From that invocation, only two possible situations can take place:

  **No node becomes stealable** In this case, the private part of $q$'s SpDeque was empty, implying some processor (either $q$ or some thief) assigned $u$.

  **A node becomes stealable** If the node that became stealable as the result of the invocation was not $u$, then either $u$ was assigned by a processor (that could have been $q$ or some thief), or $u$ had already been transfered to the public part of $q$'s SpDeque as a consequence of another thief's steal attempt that also returned EMPTY, implying that either $u$ became assigned, or it became stealable. Otherwise, the node that became stealable as a result of the *updateBottom*'s invocation was $u$. Thus, in any case, $u$ either gets assigned to a processor or becomes stealable.

With this, we conclude that $u$ either became assigned or became stealable until step $j + 2C$.

From Lemma 3.5, we have

$$\phi_i(u) \geq \frac{4}{5}\Phi_i(q).$$

Furthermore, Lemma 3.4 proves that if $u$ gets assigned the potential decreases by at least

$$\frac{3}{4}\phi_i\left(u\right),$$

and if $u$ becomes stealable the potential also decreases by at least

$$\frac{3}{4}\phi_i\left(u\right).$$

Because $u$ is either assigned or becomes stealable in any case, we conclude the potential associated with $q$ at step $j+2C$ has decreased by at least

$$\frac{4}{5}\frac{3}{4}\Phi_i\left(q\right)=\frac{3}{5}\Phi_i\left(q\right).$$

■

The following lemma is a trivial generalization of the original result presented in [8, Balls and Weighted Bins]. The only difference between the two results is the assumption of having at least $B$ balls, rather than exactly $B$ balls. Its proof is only presented for the sake of completion, and is adapted from original.

**Lemma 6.9** (Balls and Weighted Bins)**.** *Suppose we are given at least $B$ balls, and exactly $B$ bins. Each of the balls is tossed independently and uniformly at random into one of the $B$ bins, where for $i=1,\dots,B$, bin $i$ has a weight $W_i$. The total weight is $W=\sum_{i=1}^{B}W_i$. For each bin $i$, we define the random variable $X_i$ as*

$$X_i=\begin{cases}W_i & \textit{if some ball lands in bin } i\\ 0 & \textit{otherwise}\end{cases}$$

*and define the random variable $X$ as $X=\sum_{i=1}^{B}X_i$.*

*Then, for any $\beta$ in the range $0<\beta<1$, we have*

$$P\left\{X\geq\beta W\right\}\geq1-\frac{1}{\left(1-\beta\right)e}.$$

*Proof.* Consider the random variable $W_i-X_i$ taking the value of $W_i$ when no ball lands in bin $i$ and 0 otherwise, and let $B'$ denote the total number of balls that are tossed. It follows

$$\begin{aligned}E\left[W_i-X_i\right]&=W_i\left(1-\frac{1}{B}\right)^{B'}\\&\leq W_i\left(1-\frac{1}{B}\right)^{B}\\&\leq\frac{W_i}{e}\end{aligned}$$

From the linearity of expectation, we have

$$E\left[W-X\right]\leq\frac{W}{e}.$$

From Markov's Inequality it follows

$$\begin{aligned} P\left\{W - X > (1-\beta)W\right\} &\leq \frac{E\left[W - X\right]}{(1-\beta)W} \\ P\left\{X < \beta W\right\} &\leq \frac{E\left[W - X\right]}{(1-\beta)W} \\ &\leq \frac{W/e}{(1-\beta)W} \\ &= \frac{1}{(1-\beta)e} \end{aligned}$$

Thus, $P\left\{X \geq \beta W\right\} \geq 1 - \frac{1}{(1-\beta)e}$. ■

The following result states that for each $P$ idle iterations that take place, with constant probability the potential drops by a constant factor. An analogous lemma was originally presented in [8, Lemma 8] for the Non-Blocking Work Stealing algorithm. The result is a consequence of Lemmas 6.9 and 3.6 and its proof follows the same traits as the one presented in that study.

**Lemma 6.10.** *Consider any step $i$ and any later step $j$ such that at least $P$ idle iterations occur from $i$ (inclusive) to $j$ (exclusive). Then, we have*

$$P\left\{\Phi_i - \Phi_{j+2C} \geq \frac{3}{10}\Phi_i\left(D_i\right)\right\} > \frac{1}{4}.$$

*Proof.* By Lemma 3.6 we know that for each processor $p \in D_i$ that is targeted by a steal attempt, the potential drops by at least $\frac{3}{5}\Phi_i\left(p\right)$, at most $2C$ steps after being targeted.

When executing an idle iteration, a processor plays the role of a thief attempting to steal work from some victim. Thus, since $P$ idle iterations occur from step $i$ (inclusive) to step $j$ (exclusive), at least $P$ steal attempts take place during that same interval. We can think of each such steal attempt as a ball toss of Lemma 6.9.

For each processor $p$ in $D_i$, we assign it a weight

$$W_p = \frac{3}{5}\Phi_i\left(p\right),$$

and for each other processor $p$ in $A_i$, we assign it a weight

$$W_p = 0.$$

Clearly, the weights sum to

$$W = \frac{3}{5}\Phi_i\left(D_i\right).$$

Using $\beta = \frac{1}{2}$ in Lemma 6.9 it follows that with probability at least

$$1 - \frac{1}{(1-\beta)e} > \frac{1}{4},$$

the potential decreases by at least

$$\beta W = \frac{3}{10}\Phi_i\left(D_i\right),$$

concluding the proof of this lemma. ■

Finally, we bound the expected number of idle iterations that take place during a computation's execution using the LCWS algorithm. The result follows from Lemma 3.7 and is proved using the same arguments that are used in the proof of [8, Theorem 9]. The presented proof corresponds to an adaptation of the one originally presented for the just mentioned theorem.

**Lemma 6.11.** *Consider any computation with work $T_1$ and critical-path length (i.e. span) $T_\infty$ executed by LCWS using $P$ processors. The expected number of idle iterations is at most $O\left(PT_\infty\right)$. Moreover, with probability at least $1-\varepsilon$ the number of idle iterations is at most $O\left(\left(T_\infty + \ln\left(\frac{1}{\varepsilon}\right)\right)P\right)$.*

*Proof.* To analyze the number of idle iterations, we break the execution into *phases*, each composed by $\Theta\left(P\right)$ idle iterations. Then, we prove that, with constant probability, a phase leads the potential to drop by a constant factor.

A computation's execution begins when the root gets assigned to a processor. By definition, the weight of the root is $T_\infty$, implying the potential at the beginning of a computation's execution starts at $\Phi_0 = 4^{3T_\infty - 2}$. Further, it is straightforward to deduce that the potential is 0 after (and only after) a computation's execution terminates. We use these facts to bound the expected number of phases needed to decrease the potential down to 0. The first phase starts at step $t_1 = 1$, and ends at the first step $t_1{}'$ such that, at least $P$ idle iterations took place during the interval $[t_1, t_1{}' - 2C]$. The second phase starts at step $t_2 = t_1{}' + 1$, and so on.

Consider two consecutive phases starting at steps $i$ and $j$ respectively. We now prove that

$$P\left\{\Phi_j \leq \frac{7}{10}\Phi_i\right\} > \frac{1}{4}.$$

Recall that we can partition the potential as

$$\Phi_i = \Phi_i\left(A_i\right) + \Phi_i\left(D_i\right).$$

Since, from the beginning of each phase and until its last $2C$ steps, at least $P$ idle iterations take place, then, by Lemma 3.7 it follows

$$P\left\{\Phi_i - \Phi_j \geq \frac{3}{10}\Phi_i\left(D_i\right)\right\} > \frac{1}{4}.$$

Now, we have to prove the potential also drops by a constant fraction of $\Phi_i\left(A_i\right)$. Consider some processor $p \in A_i$:

- If $p$ does not have an assigned node, then

$$\Phi_i\left(p\right) = 0.$$

- Otherwise, if $p$ has an assigned node $u$ at step $i$, then,

$$\Phi_i\left(p\right) = \phi_i\left(u\right).$$

  Noting that each phase has more than $C$ steps, then, $p$ executes $u$ before the next phase begins (*i.e.* before step $j$). Thus, the potential drops by at least $\frac{47}{64}\phi_i\left(u\right)$ during that phase.

Cumulatively, for each $p \in A_i$, it follows

$$\Phi_i - \Phi_j \geq \frac{47}{64}\Phi_i\left(A_i\right).$$

Thus, no matter how $\Phi_i$ is partitioned between $\Phi_i(A_i)$ and $\Phi_i(D_i)$, we have

$$P\left\{\Phi_i - \Phi_j \geq \frac{3}{10}\Phi_i\right\} > \frac{1}{4}.$$

We say a phase is successful if it leads the potential to decrease by at least a $\frac{3}{10}$ fraction. So, a phase succeeds with probability at least $\frac{1}{4}$. Since the potential is an integer, and, as aforementioned, starts at $\Phi_0 = 4^{3T_\infty - 2}$ and ends at 0, then, there can be at most

$$\begin{aligned}(3T_\infty - 2)\log_{\frac{10}{7}}(4) &= (3T_\infty - 2)\frac{\ln(4)}{\ln\left(\frac{10}{7}\right)} \\ &< 12T_\infty\end{aligned}$$

successful phases. If we think of each phase as a coin toss, where the probability that we get heads is at least $\frac{1}{4}$, then, the expected number of coins we have to toss to get heads $12T_\infty$ times is at most $48T_\infty$. In the same way, the expected number of phases needed to obtain $12T_\infty$ successful ones is at most $48T_\infty$. Consequently, the expected number of phases is $O(T_\infty)$. Moreover, as each phase contains $O(P)$ idle iterations, the expected number of idle iterations is $O(PT_\infty)$.

Now, suppose the execution takes $n = 48T_\infty + m$ phases. Each phase succeeds with probability greater or equal to $p = \frac{1}{4}$, meaning the expected number of successes is at least

$$np = 12T_\infty + \frac{m}{4}.$$

We now compute the probability that the number of $X$ successes is less than $12T_\infty$. We use *Chernoff bound* [6],

$$P\{X < np - a\} < e^{-\frac{a^2}{2np}}$$

with $a = \frac{m}{4}$. It follows,

$$np - a = 12T_\infty + \frac{m}{4} - \frac{m}{4} = 12T_\infty.$$

Choosing $m = 48T_\infty + 16\ln\left(\frac{1}{\varepsilon}\right)$, We have

$$\begin{aligned}P\{X < 12T_\infty\} &< e^{-\frac{\left(\frac{m}{4}\right)^2}{2\left(12T_\infty + \frac{m}{4}\right)}} \\ &= e^{-\frac{\left(\frac{m}{4}\right)^2}{24T_\infty + \frac{m}{2}}} \\ &\leq e^{-\frac{\left(\frac{m}{4}\right)^2}{\frac{m}{2} + \frac{m}{2}}} \\ &= e^{-\frac{(m/4)^2}{m}} \\ &= e^{-\frac{m}{16}} \\ &\leq e^{-\frac{16\ln\left(\frac{1}{\varepsilon}\right)}{16}} \\ &= e^{\ln(\varepsilon)} \\ &= \varepsilon\end{aligned}$$

Thus, the probability that the execution takes $96T_\infty + 16\ln\left(\frac{1}{\varepsilon}\right)$ phases or more, is less than $\varepsilon$. With this we conclude that the number of idle iterations is at most

$$O\left(\left(T_\infty + \ln\left(\frac{1}{\varepsilon}\right)\right)P\right)$$

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