A communication efficient Work Stealing and Spreading scheduler for structured parallel computations

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

Algorithms for scheduling structured parallel computations have been widely studied in the literature. For some time now, Work Stealing is one of the most popular for scheduling such computations, and its performance has been studied in both theory and practice. Unfortunately, even though the expected execution time of a computation using Work Stealing is asymptotically optimal, the performance of the scheduler was proved to be limited. It was recently proved that the Greedy Work Stealing and Spreading scheduler — a variant of the Work Stealing scheduler where, in addition to stealing, processors are allowed to spread work as it is generated — overcomes the performance limitations of Work Stealing. Nevertheless, this new algorithm suffers from an even more severe drawback that is due to the great amount of extra communication that the spreading mechanism incurs. In this paper we introduce a Work Stealing and Spreading algorithm that provably overcomes the limitation of Work Stealing while maintaining its communication efficiency. To mitigate the high communication costs that are typical of schedulers that rely on spreading, we only permit processors to make at most one spread attempt for each steal attempt that takes place, thus limiting the number of spread attempts to be at most the number of steal attempts and allowing to amortize the communication costs of spreading with the ones of stealing. As such, our algorithm overcomes the performance limitation of WS while maintaining its low, and, in fact, asymptotically optimal communication costs.

1 Introduction

The main goal of a structured computation's scheduler is to guarantee the fast completion of a computation's execution. These computations are usually modelled as single-source single-sink direct acyclic graphs (dags) — whose nodes denote instructions and whose edges denote ordering constraints between instructions — where each node's out-degree is at most two. For some time now, Work Stealing is considered the *de facto* standard for scheduling structured computations [1, 2, 3, 5, 10, 14, 15, 18, 19, 20, 22]. In Work Stealing (or WS for short), each processor owns a deque that uses to keep track of its work. Busy processors operate locally on their deques, adding and retrieving work from them as necessary, until they run out of work. When that happens, a processor becomes a thief and starts a stealing phase, during which it targets other processors, uniformly at random, in order to steal work from their deques. As proved in [5, 10], the expected execution time of any computation using WS is optimal up to a constant factor. Yet, WS's performance is known to be limited when executing computations that exhibit unbalanced parallelism [2, 14, 15]. As a matter of fact, it has recently been proved that there are computations during whose execution (using WS) the expected amount of work executed by processors is surpassed by the amount of work generated, regardless of the ratio of processors that are idle, confirming WS's performance limitation [**?**]. In that same work, the authors studied a variant of WS where, in addition to stealing, processors are allowed to spread work as it is generated, and verified that the new scheduler overcomes the limitations of WS. Nevertheless, and as pointed out by the authors, since, under that algorithm, processors always attempt spreading work when they have a chance, the scheduler incurs severe extra communication costs that are, for practical purposes, counterproductive, and so, the problem of coming up with a communication efficient scheduler that overcomes WS's limitations remains open.

To this extent, the central contribution of this paper lies in the presentation of a communication efficient Work Stealing and Spreading scheduler that overcomes WS's limitations (Section 3). The main idea behind the Work Stealing and Spreading algorithm is, on one hand, to overcome the performance limitation of WS by allowing processors to spread work as it is generated, and, on the other hand, to avoid unnecessary communication between processors, keeping the total communication costs of the scheduler up to a constant factor away from the communication costs of WS, which, in turn, are known to be asymptotically optimal [10]. For disambiguation, we refer to the Work Stealing and Spreading algorithm given in [**?**] as Greedy Work Stealing and Spreading (or GWSS for short) and to the one we present in this paper as Work Stealing and Spreading, or just WSS for short. The key features of WSS are:

1. Its ability to overcome the performance limitation of WS. To that end, we prove that WSS is *algorithm short-term stable wrt* (with respect to) $[0.8675; 1[$ (Section 4), meaning that, if at some moment during a computation's execution the ratio of idle processors is at least 0.8675, then the amount of work executed by processors is expected to surpass the maximum possible amount of work that may be generated.
2. Its ability to overcome the communication deficiency of GWSS. While in GWSS a processor makes spread attempts whenever it has a chance, in WSS a processor is only allowed to make a spread attempt after being targeted by a thief. Since WS is extremely communication efficient [10, 12], by permitting at most one spread attempt for each steal attempt we guarantee that WSS is communication efficient.

2 Preliminaries

In this section we present the formal background used for analyzing WSS. All the definitions, results, assumptions, *etc*, given in this section were originally introduced in [**?**].

As already mentioned, like in much previous work [1, 3, 5, 10, 20, 22] we model a structured computation as a *dag* $G = (V, E)$, where each node $v \in V$ corresponds to an instruction, and each edge $(\mu_1, \mu_2) \in E$ denotes an ordering constraint (meaning μ_2 can only be executed after μ_1). Nodes with in-degree of 0 are referred to as *roots*, while nodes with out-degree of 0 are called *sinks*. We make two standard assumptions related with the structure of computations. Let G denote a computation's dag: 1. there is only one root and one sink in G; and 2. the out-degree of any node within G is at most two. In addition, we consider that the execution of computations is carried out by a set of processors denoted by $Procs$ whose cardinality is denoted by P. We assume that $P \geq 2$ (*i.e.* $Procs$ contains at least two processors), and that all processors operate synchronously in time steps. Moreover, we consider that processors operate on discrete time steps, each executing one instruction — that may or may not correspond to a node — per time step. Therefore a computation's execution can be partitioned into discrete time steps *s.t.* (such that) at each step every processor executes an instruction. We refer to these time steps using non-negative integers, where 0 is the first step.

Definition 2.1. At any step during a computation's execution each node is in one of the following states: 1. **not ready** — if its ancestors have not yet been executed; 2. **ready** — if its ancestors have been executed, but not the node itself; and 3. **executed** — if the node has been executed.

To guarantee correctness, only nodes that are **not ready** can become **ready**, and only nodes that are **ready** can become **executed**. For each step i, refer to the set of nodes that are: 1. **not ready** by $NonReady_i$; 2. **ready** by $Ready_i$ (or simply R_i); and 3. **executed** by $Executed_i$. Say that a node is *enabled* at step i if it was **not ready** at step i but is **ready** at step $i+1$, and, similarly, that a node is *executed* (or *computed*) at step i if it was **ready** at step i but is **executed** at step $i+1$. Additionally, for each step i, we partition R_i into P sets (one per processor) and for each $p \in Procs$, we denote the set of nodes that are *attached* to p at step i by $R_i(p)$ — p's partition of R_i. A node μ is *migrated* if $\mu \in R_i\,(p)$ and $\mu \in R_{i+1}\,(q)$, where $p \neq q$, for $p, q \in Procs$.

Definition 2.2. For each step i and $p \in Procs$, define the set of nodes *enabled* by p as $E_i(p) = R_{i+1}(p) - R_i$, and *executed* by p as $C_i(p) = R_i(p) - R_{i+1}$. Define the set of nodes *migrated* from p to all other processors as $M_i^+(p) = R_i(p) \cap (R_{i+1} - R_{i+1}(p))$ and from all other processors to p as $M_i^-(p) = R_{i+1}(p) \cap (R_i - R_i(p))$. For a set of processors $S \in \mathcal{P}(Procs)$, define $R_i(S) = \bigcup_{p \in S} R_i(p)$, $E_i(S) = \bigcup_{p \in S} E_i(p)$, $C_i(S) = \bigcup_{p \in S} C_i(p)$, $M_i^+(S) = \bigcup_{p \in S} M_i^+(p)$, and $M_i^-(S) = \bigcup_{p \in S} M_i^-(p)$. Finally, define $E_i = E_i(Procs)$, $C_i = C_i(S)$, $M_i^+ = M_i^+(Procs)$, and $M_i^- = M_i^-(Procs)$.

Definition 2.3. A *round* is a sequence of L time steps (for some constant $L \geq 1$) *s.t.* a computation's execution can be partitioned into equal-length rounds and for every round: 1. no processor executes more than a single node; and 2. no node is migrated more than once.

Analogously to time steps, refer to rounds using non-negative integers, but with an additional bar, where $\overline{0}$ denotes the first round. Throughout this paper, we let L denote the length of rounds and $\overline{t}\,[i]$ denote the *i-th* step of some round $\overline{t}$ (for $i \in \{0, \ldots, L-1\}$).

Definition 2.4. For each round $\overline{t}$ and processor p, define the set of nodes *attached* to p at round $\overline{t}$ as $R_{\overline{t}}(p) = R_{\overline{t}[0]}(p)$, *enabled* by p during $\overline{t}$ as $E_{\overline{t}}(p) = \bigcup_{i \in \{\overline{t}[0], \cdots, \overline{t}[L-1]\}} E_i(p)$, *executed* by p during $\overline{t}$ as $C_{\overline{t}}(p) = \bigcup_{i \in \{\overline{t}[0], \cdots, \overline{t}[L-1]\}} C_i(p)$, *migrated* from p to all other processors during $\overline{t}$ as $M_{\overline{t}}^+(p) = \bigcup_{i \in \{\overline{t}[0], \cdots, \overline{t}[L-1]\}} M_i^+(p)$ and *migrated* from all other processors to p during $\overline{t}$ as $M_{\overline{t}}^-(p) = \bigcup_{i \in \{\overline{t}[0], \cdots, \overline{t}[L-1]\}} M_i^-(p)$. For a set of processors $S \in \mathcal{P}(Procs)$, define $R_{\overline{t}}(S) = \bigcup_{p \in S} R_{\overline{t}}(p)$, $E_{\overline{t}}(S) = \bigcup_{p \in S} E_{\overline{t}}(p)$, $C_{\overline{t}}(S) = \bigcup_{p \in S} C_{\overline{t}}(p)$, $M_{\overline{t}}^+(S) = \bigcup_{p \in S} M_{\overline{t}}^+(p)$, and $M_{\overline{t}}^-(S) = \bigcup_{p \in S} M_{\overline{t}}^-(p)$. Finally, define $E_{\overline{t}} = E_{\overline{t}}(Procs)$, $C_{\overline{t}} = C_{\overline{t}}(S)$, $M_{\overline{t}}^+ = M_{\overline{t}}^+(Procs)$, and $M_{\overline{t}}^- = M_{\overline{t}}^-(Procs)$.

The following requirement gives us the guarantee that a processor p only executes a node μ during a round $\overline{t}$ if μ is attached to p at the beginning of that round.

Requirement 2.5. For any round $\overline{t}$ and $p \in Procs$, we must have $R_{\overline{t}}(p) \supseteq C_{\overline{t}}(p)$.

Definition 2.6. Say that a processor $p \in Procs$ is *idle* during a round $\overline{t}$ if $C_{\overline{t}}(p) = \emptyset$, and, otherwise, say that p is *busy*. Moreover, denote the number of idle processors during a round $\overline{t}$ by $P_{\overline{t}}^{idle}$, and define $\alpha_{\overline{t}}$ as the ratio of idle processors, $\alpha_{\overline{t}} = P_{\overline{t}}^{idle}/P$.

For each round $\overline{t}$, we classify processors according to whether they execute all their attached nodes during $\overline{t}$ or not. If a processor p executes all its attached nodes during $\overline{t}$ then p is *self-stable*, and, otherwise, p is *non-self-stable*.

Definition 2.7. Define the set of *self-stable* and *non-self-stable* processors at some round $\overline{t}$ as $S_{\overline{t}} = \{p \in Procs \mid R_{\overline{t}}(p) = C_{\overline{t}}(p)\}$ and $U_{\overline{t}} = Procs - S_{\overline{t}}$, *resp* (respectively).

Now, we present the criterion for measuring computation schedulers' performance: *algorithm short-term stability.* Intuitively, a scheduler is *algorithm short-term stable wrt* (with respect to) an interval $I \subseteq]0;1[$ *iff* (if and only if) for any round $\overline{t}$ *s.t.* $\alpha_{\overline{t}} \in I$, the amount of work attached to non-self stable processors that is not executed is expected to decrease, while the amount of work attached to each self-stable processor does not grow unboundedly.

Definition 2.8 (Algorithm short-term stability)**.** A scheduling algorithm is *algorithm short-term stable* with respect to an interval $I \subseteq]0;1[$, *iff* for any round $\overline{t}$,

$$(\alpha_{\overline{t}} \in I) \Rightarrow \left[\left(\mathsf{E}[|R_{\overline{t+1}}(U_{\overline{t}}) - C_{\overline{t+1}}(U_{\overline{t}})|] < |R_{\overline{t}}(U_{\overline{t}}) - C_{\overline{t}}(U_{\overline{t}})|\right) \wedge \left(\forall p \in S_{\overline{t}}, \left|R_{\overline{t+1}}(p)\right| \leq L+1\right)\right],$$

where L denotes the length of the rounds.

The reason for limiting the number of nodes that can become attached to each self-stable processor during a round is to disallow schedulers to keep ping-ponging work between non-self-stable and self-stable processors throughout the execution[1].

The following lemma relates the performance of each individual non-self-stable processor with the performance of all non-self-stable processors $U_{\overline{t}}$. See [?, Lemma 2.10] for its proof.

Lemma 2.9. *For any round* $\overline{t}$*, if* $\forall p \in U_{\overline{t}}$, $\mathsf{E}[\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right|] < |R_{\overline{t}}(p) - C_{\overline{t}}(p)|$*, then we have* $\mathsf{E}[\left|R_{\overline{t+1}}(U_{\overline{t}}) - C_{\overline{t+1}}(U_{\overline{t}})\right|] < |R_{\overline{t}}(U_{\overline{t}}) - C_{\overline{t}}(U_{\overline{t}})|$.

As we will see in Section 4, under WSS, if a processor $p \in Procs$ is non-self-stable during some round $\overline{t}$, then $M_{\overline{t}}^{-}(p) = \emptyset$. Thus, the following Corollary is key for WSS's analysis because it relates, for each round $\overline{t}$, the difference in the number of nodes that a processor p enables during $\overline{t}$, that are migrated from p during $\overline{t}$, and that p executes during $\overline{t+1}$, with the *algorithm short-term stability* of WSS. See [?, Lemma 2.13] for its proof.

Corollary 2.10. *For any round* $\overline{t}$*, if* $(p \in U_{\overline{t}}) \Rightarrow \left(M_{\overline{t}}^{-}(p) = \emptyset\right)$*, then* $|E_{\overline{t}}(p)| < \left|C_{\overline{t+1}}(p)\right| + \left|M_{\overline{t}}^{+}(p)\right|$ *iff* $\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right| < |R_{\overline{t}}(p) - C_{\overline{t}}(p)|$.

2.1 The methodology

We now recapitulate a methodology for analyzing the performance of randomized online structured computation schedulers, originally presented in [?], that will be used to analyze WSS. We only outline the essential definitions and assumptions of this methodology that are necessary for WSS's analysis — specifically, the ones that are required to permit ordering the actions that processors take during each round.

To begin, we require that the scheduling algorithm to be analyzed must be definable by a cycle *s.t.*: 1. at most one of the instructions composing any particular iteration of this cycle may correspond to a node's execution; 2. no node that is migrated to a processor p, who is executing an iteration of this cycle, can be migrated again (to another processor), before p finishes the current

[1]For a more careful description of *algorithm short-term stability*, please refer to [?].

iteration; 3. the length of any sequence of instructions that corresponds to some execution of this cycle is at most constant; and 4. the full sequence of instructions executed by any processor can be partitioned into smaller sub-sequences, each corresponding to a particular execution of this cycle. We refer to this cycle as the *scheduling loop*, and to any sequence of instructions that correspond to some iteration of a scheduling loop as *scheduling iteration.* To order the actions that processors take during scheduling iterations, each iteration can be partitioned into a sequence of *phases.* The way iterations are partitioned depends on the algorithm being studied. For example, as we will see in 3, each scheduling iteration of the WSS algorithm is partitioned into three phases.

Although, with the introduction of phases, it is possible to order the actions that processors take during the execution of every iteration, it is still not possible to order the actions that processors take during each round — while scheduling iterations (and phases) are related with the sequences of instructions that processors execute, rounds are related with time. To meet our needs we have to guarantee that all processors start the execution of each phase at the same time step, and that the execution of each iteration matches with a round. Our first step towards meeting our ordering needs is to make the assumption that all processors begin working at the same time. Refer to the step at which a processor p executes its i-th instruction as $\chi(p, i)$.

Assumption 2.11. $\forall p \in Procs, \quad \chi(p, 1) = 0.$

Now, we present the *synch* procedure, which allows to synchronize processors at the end of each phase. The *synch* procedure takes two input parameters: 1. $maxPhaseLength$ — the length of a longest sequence of instructions that may compose a given phase, and; 2. $currentPhaseLength$ — the number of time steps during which the processor has been executing the current phase, preceding the procedure's invocation. Given these parameters, *synch* adds a sequence of $maxPhaseLength - currentPhaseLength$ no-op instructions, guaranteeing that the number of time steps taken from the beginning of each phase's execution until the end of the *synch* procedure's call is the same for all processors. To use the *synch* procedure, we rely on the purely theoretical function ι to obtain the value of $currentPhaseLength$.

Although we do not delve into further details here, note that we are able to keep processors synchronized, on a phase basis, throughout any computation's execution. Thus, to analyze a scheduler's performance (using the methodology presented in [**?**]) it suffices to: 1. define the scheduler by a scheduling loop; 2. divide the actions that processors take during each iteration of the loop (by partitioning each scheduling iteration into phases); and 3. insert a call to the *synch* procedure at the end of each phase.

3 The Work Stealing and Spreading algorithm

As already mentioned, WSS (depicted in Algorithm 1), is a variant of the GWSS scheduler given in [**?**]. Informally, while in GWSS processors make spread attempts whenever they have a chance, in WSS, processors may only perform spread attempts after being *"solicited"* by an idle processor (more on this ahead). Similarly to WS and GWSS, in WSS each processor owns a constant-time lock-free deque object that supports three methods: *pushBottom*, *popBottom* and *popTop*. Only the owner of a deque may invoke the *pushBottom* and *popBottom* methods, which, respectively, add a node to the bottom of the deque, and remove and return the bottommost node of the deque, if any. The *popTop* method is invoked by processors searching for work, and for each invocation to this method, the deque's current topmost node is guaranteed to be removed and returned, either by such invocation or by some concurrent one[2]. In addition to the deque, each processor has a variable *assigned* that stores the node that it will execute next, if any. To implement the spreading

[2]For a more careful description of the lock-free deque semantics, originally defined in [5], please refer to Section A.

mechanism each processor additionally owns a *state* flag, a *donation* cell and a *spreading* flag. Processors use the *state* flag to inform other processors on their current state — WORKING, IDLE or marked as target of a donation (more on this ahead) —, the *donation* cell to store nodes that they want to spread, and the *spreading* flag to decide whether to make a spread attempt when there is a chance or not. Before a computation's execution begins, each processor has its deque empty, its *assigned* node set to NONE, its *state* flag set to IDLE, its *donation* cell with the value of NONE and its *spreading* flag set to FALSE. To begin the execution, one of the processors gets the *root* node assigned and has its *state* flag set to WORKING.

We assume that processors are uniquely identified by an *id*, with which they can be accessed in constant time. The scheduler also makes use of the *CAS* instruction (Compare-And-Swap), with its usual semantics. Thus, at most one *CAS* instruction targeting the same memory location can successfully execute at each step. We assume that the processor that succeeds executing the *CAS* instruction over a memory address m at some step i is chosen uniformly at random from the set of processors that are eligible to successfully execute the instruction at step i over memory address m.

Algorithm 1 WSS — Part 1.

```
1: procedure SCHEDULER( )
2:   while not finished (computation) do
3:     if ValidNode (self.assigned) then
4:       enabled ← execute (self.assigned)
5:       assigned ← NONE
6:       synch($max\_phase_{I}\_length$, $\iota()$)
7:       if length (enabled) > 0 then
8:         self.assigned ← enabled[0]
9:         if length (enabled) = 2 then
10:          self.handleExtraNode (enabled[1])
11:        else
12:          synch($max\_phase_{II}\_length$, $\iota()$)
13:        end if
14:      else
15:        synch($max\_phase_{II}\_length$, $\iota()$)
16:        self.assigned ← self.deque.popBottom ()
17:        if not ValidNode (self.assigned) then
18:          self.state ← IDLE
19:        end if
20:      end if
21:    else
22:      self.loadBalance ()
23:    end if
24:    synch($max\_phase_{III}\_length$, $\iota()$)
25:  end while
26: end procedure

27: function VALIDNODE(node)
28:   return node ≠ EMPTY
29:       and node ≠ ABORT
30:       and node ≠ NONE
31: end function
```

Algorithm 2 WSS — Part 2.

```
32: procedure HANDLEEXTRANODE($\mu$)
33:   result ← FAILURE
34:   if self.spreading then
35:     self.donation ← $\mu$
36:     donee ← UniformlyRandomProcessor()
37:     result ← CAS (donee.state, IDLE, self.id)
38:   end if
39:   synch($max\_phase_{II}\_length$, $\iota()$)
40:   self.spreading ← FALSE
41:   if result ≠ SUCCESS then
42:     self.deque.pushBottom ($\mu$)
43:   end if
44: end procedure

45: procedure LOADBALANCE( )
46:   victim ← UniformlyRandomProcessor()
47:   self.assigned ← victim.deque.popTop ()
48:   if ValidNode (self.assigned) then
49:     self.state ← WORKING
50:   end if
51:   processorToUpdate ← UniformlyRandomProcessor()
52:   processorToUpdate.spreading ← TRUE
53:   synch($max\_phase_{I}\_length$, $\iota()$)
54:   synch($max\_phase_{II}\_length$, $\iota()$)
55:   if self.state ≠ IDLE and self.state ≠ WORKING then
56:     donor ← processor [self.state]
57:     self.assigned ← donor.donation
58:     self.state ← WORKING
59:   end if
60: end procedure
```

With the employment of the *spreading* flag, we can guarantee that the total number of spread attempts does not exceed the total number of steal attempts. As such, and in contrast with GWSS, WSS avoids most of the unnecessary communication that are typical of work spreading strategies. More, assuming that every steal attempt and every spread attempt incur at most a constant amount of communication, the expected total communication of WSS is a constant factor away from the expected total communication of the WS scheduler (as given in [10]), which, in turn, was proved to be asymptotically optimal [10, 12]. We do not give a formal proof of this claim because the arguments given in [10] can be trivially adapted for WSS, by amortizing the costs of each spread attempt with the costs of the corresponding steal attempt.

3.1 Preparing for the analysis of WSS's performance

We now begin the preparation for the performance analysis of WSS. As one can observe from Algorithm 1 (and noting that the deque implementation is, as already mentioned, constant-time), WSS can be naturally defined by a scheduling loop (lines 2 to 25): 1. at most one of the instructions within the sequence of a scheduling iteration corresponds to the execution of a node (line 4); 2. no node that is migrated to a processor is migrated ever again, as it becomes the processor's new assigned node (lines 47 and 57); 3. the length of any iteration of the scheduling loop is bounded by a constant; and 4. the full sequence of instructions executed by any processor can be partitioned into scheduling iterations.

We partition each iteration of WSS into a sequence of three phases that we now briefly describe:

Phase I If a processor p has a valid assigned node, it executes the node (line 4). Otherwise, p attempts to load balance. To that end: 1. p targets a victim processor uniformly at random (line 46), and invokes the *popTop* method to the victim's deque, in an attempt to steal a node (line 47). If the attempt succeeds the stolen node becomes p's new assigned node. 2. after attempting a steal, and regardless of its outcome, p targets, uniformly at random, another processor q (line 51) and sets q's *spreading* flag to TRUE (line 52).

Phase II If, in phase I, a processor p made a steal attempt or executed a node that did not enable any node, then p does not take any action during this phase. Otherwise, if at least one node was enabled, one of the enabled nodes becomes p's new assigned node (line 8). If two nodes were enabled, then, after having a new node assigned and depending on the state of its *spreading* flag, p may attempt to spread the node it did not assign (lines 34 to 38).

Phase III If a processor p executed a node in the first phase but no node was enabled, p invokes *popBottom* to fetch the bottom-most node from its deque, if there is any. On the other hand, if a single node was enabled, p does not take any action during this phase. If two nodes were enabled, p only takes action if it did not perform a successful donation attempt during the second phase (*i.e.* if p did not perform a donation attempt at all, or if p actually performed a donation attempt, but the attempt was not successful). In this situation, p pushes the node it did not assign into the bottom of its own deque, via the *pushBottom* method (line 42). Finally, if the processor made an unsuccessful steal attempt during the first phase, it polls its *state* flag checking for incoming donations (line 55). If there is one, p transfers the node from the donor's *donation* cell (line 57).

The WSS scheduler is depicted in Algorithm 1, where $max\_phase_{I}\_length$, $max\_phase_{II}\_length$ and $max\_phase_{III}\_length$ are constants that correspond to the lengths of the longest sequences of instructions composing each of the phases of WSS. Thus, to proceed to analysis of WSS's performance, it only remains to show that WSS satisfies Requirement 2.5. At the beginning of any round, each node that is attached to a processor is either in its deque or is the processor's currently assigned node. As it can be observed in Algorithm 1, a processor only executes the node that is stored in its *assigned* variable. Since the value of this variable is not changed at least until the processor executes the node, then the node was already stored in the *assigned* variable when the round began, and so the requirement is satisfied.

4 Analysis of Work Stealing and Spreading

In this section we analyze the performance of the Work Stealing and Spreading algorithm, showing that, as GWSS, it successfully overcomes the limitations of WS identified in [**?**]. The analysis consists on proving that the scheduler is *algorithm short-term stable wrt* $[0.8675; 1[$, and the outlines of the proof we present are similar to the ones given in [**?**] for the proof of GWSS's *algorithm short-term stability*[3].

Theorem 4.1. *WSS (as defined in Algorithm 1) is algorithm short-term stable wrt* $[0.8675; 1[$.

Before beginning with the proof of this result, we introduce two more definitions, originally presented in [**?**], that are key to distinguish which nodes are migrated due to stealing and which nodes are migrated due to spreading.

Definition 4.2. Refer to the set of nodes stolen at step i from a processor p as $Stolen_i^+(p)$, and to the set of nodes stolen by p as $Stolen_i^-(p)$. Moreover, for some round $\bar{t}$, define the set of nodes stolen during $\bar{t}$ from p as $Stolen_{\bar{t}}^+(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[L-1]\}} Stolen_i^+(p)$, and the set of nodes stolen by p as $Stolen_{\bar{t}}^-(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[L-1]\}} Stolen_i^-(p)$.

Definition 4.3. Refer to the set of nodes spread at step i by a processor p as $Spread_i^+(p)$, and to the set of nodes spread to p as $Spread_i^-(p)$. Additionally, for some round $\bar{t}$, define the set of nodes spread during $\bar{t}$ by p as $Spread_{\bar{t}}^+(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[L-1]\}} Spread_i^+(p)$, and the set of nodes spread to p as $Spread_{\bar{t}}^-(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[L-1]\}} Spread_i^-(p)$.

The three following results and their corresponding proofs are equivalent to [**?**, Lemmas *D.1*, *D.2* and *D.3*]. Since the proofs follow from the definition of WSS, or can be trivially adapted for WSS, we omit them.

Lemma 4.4. *For any* $p \in Procs$ *and round* $\bar{t}$ *during a computation's execution by WSS,* $M_{\bar{t}}^+(p) = Stolen_{\bar{t}}^+(p) \cup Spread_{\bar{t}}^+(p)$ *and* $M_{\bar{t}}^-(p) = Stolen_{\bar{t}}^-(p) \cup Spread_{\bar{t}}^-(p)$.

Lemma 4.5. *For any* $p \in Procs$ *and round* $\bar{t}$ *during a computation's execution by WSS,* $Stolen_{\bar{t}}^+(p) \cap Spread_{\bar{t}}^+(p) = \emptyset$.

Lemma 4.6. *Consider some* $p \in Procs$ *and some round* $\bar{t}$ *during the execution of a computation by WSS. Then: 1. if* $p \in U_{\bar{t}}$ *then* p*'s deque is non-empty and* $M_{\bar{t}}^-(p) = \emptyset$*; and 2. if* $p \in S_{\bar{t}}$ *then* $\left|R_{\overline{t+1}}(p)\right| \leq 2$.

Recall that L denotes the length of each round, and that by definition $L \geq 1$. From Lemma 4.6, it follows that $\forall p \in S_{\bar{t}}, \left|R_{\overline{t+1}}(p)\right| \leq 2 \leq L + 1$. Thus, taking into account Lemma 2.9 and Corollary 2.10, to prove this theorem, it is *STP* that for any round $\bar{t}$ *s.t.* $\alpha_{\bar{t}} \in [0.8675; 1[$, we have $\forall p \in U_{\bar{t}}, \quad |E_{\bar{t}}(p)| < \mathsf{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M_{\bar{t}}^+(p)\right|]$.

Lemma 4.7. *Suppose there are* B *bins and* $B.\alpha$ *balls, and that each ball is tossed independently and uniformly at random into the bins. For each bin* b_i*, let* Y_i *be an indicator variable, defined as*

$$Y_i = \begin{cases} 1 & \textit{if at least one ball lands in } b_i; \\ 0 & \textit{otherwise.} \end{cases}$$

Then, $\mathsf{E}[Y_i] = \mathsf{P}\{Y_i = 1\} \geq 1 - e^{-\alpha}$.

Proof of Lemma 4.7. The probability that no ball lands in b_i is $\mathsf{P}\{Y_i = 0\} = \left(1 - \frac{1}{B}\right)^{B\alpha} \leq e^{-\alpha}$. To conclude, $\mathsf{E}[Y_i] = \mathsf{P}\{Y_i = 1\} \geq 1 - e^{-\alpha}$. ■

Lemma 4.8. *For any round* $\bar{t}$ *and* $p \in U_{\bar{t}}$ *during a computation's execution using WSS,* $1 - e^{-\alpha_{\bar{t}}} \leq \mathsf{E}[|Stolen_{\bar{t}}^+(p)|] \leq \alpha_{\bar{t}}$.

Proof of Lemma 4.8. By observing Algorithm 1, it follows that a processor makes a steal attempt *iff* it is idle, implying that exactly $P\alpha_{\bar{t}}$ steal attempts are made during round $\bar{t}$. Note that: 1. steal attempts are independent from one another; and 2. a steal attempt corresponds to targeting a

[3] Although it may seem like it, in our opinion, the proof we give is not a trivial extension of the one obtained for GWSS, as studying the spreading mechanism of WSS is much more complex than studying the one of WSS where processors always make a spread attempt whenever they have the chance.

processor uniformly at random and then invoking the *popTop* method to its deque. If we imagine that each steal attempt is a ball toss and that each processor's deque is a bin, it follows by Lemma 4.7 that the probability of p's deque being targeted is at least $1-e^{-\alpha_{\bar{t}}}$. On the other hand, the expected number of invocations to the *popTop* method of any processor p's deque is $\left(P\alpha_{\bar{t}}\right)/P = \alpha_{\bar{t}}$. Since p may only invoke the *popBottom* method of its deque during the third phase and the all the steal attempts take place during the first phase, then, taking into account the deque semantics (see Section A): 1. if p's deque is targeted by at least one steal attempt, then at least one node is stolen; and 2. at most one node might be returned for each invocation to the *popTop* method. Thus, $\mathsf{E}[|Stolen^{+}_{\bar{t}}(p)|] \geq 1-e^{-\alpha_{\bar{t}}}$ and $\mathsf{E}[|Stolen^{+}_{\bar{t}}(p)|] \leq \alpha_{\bar{t}}$. ■

Consider any processor $p \in U_{\bar{t}}$. Due to our conventions related with computations' structure, and, in particular, with the node's out-degree assumption, $|E_{\bar{t}}(p)|$ is either 0, 1, or 2:

- If $|E_{\bar{t}}(p)| = 0$ then, by Lemma 4.8 it follows $|E_{\bar{t}}(p)| < \mathsf{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\bar{t}}(p)\right|]$.
- If $|E_{\bar{t}}(p)| = 1$ then, by the definition of WSS it follows $\left|C_{\overline{t+1}}(p)\right| = 1$. Taking into account Lemma 4.8, we deduce $|E_{\bar{t}}(p)| < \mathsf{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\bar{t}}(p)\right|]$.
- By the definition of WSS it follows that if $|E_{\bar{t}}(p)| = 2$ then $\left|C_{\overline{t+1}}(p)\right| = 1$. Thus, to prove this case it is *STP* that $1 < \mathsf{E}[\left|M^{+}_{\bar{t}}(p)\right|]$. We now prove just that. Throughout the rest of the analysis, assume that p is a processor that enables two nodes and that we are referring to the WSS scheduler, depicted in Algorithm 1.

By the definition of the scheduler, since p enables 2 nodes, the processor attempts to spread the node it did not assign during the second phase. As one might note, by the semantics of the spreading mechanism, the only processors that are available for receiving a node donation are the ones who are idle and whose steal attempt (that took place during the first phase) failed, and each such processor may only accept at most a single node donation.

We now prove that, the greater is the number of processors making spread attempts, the smaller are the chances that p's spread attempts succeeds.

Lemma 4.9. *Let $spreads(p, \alpha, d)$ be a function corresponding to the expected number of nodes that p spreads during any round for which the ratio of idle processors is α, and the number of processors enabling two nodes is d, under WSS. Then, $spreads(p, \alpha, d) \geq spreads(p, \alpha, P(1-\alpha))$.*

Proof of Lemma 4.9. Let B denote the set of processors that are busy during the round, G the set of processors that enabled two nodes during the round (G is a subset of B), and S the set of processors that make a spread attempt (S is a subset of G). Now, consider a scenario (for the same round) that is completely equivalent to the original, except that in this new scenario every busy processor enables two nodes. Let B', G', S' denote the sets of processors that, in the new scenario, are busy, enabled two nodes and made a spread attempt, respectively. Additionally, let d' denote the number of processors enabling two nodes under the new scenario. As for the original scenario, $B' \supseteq G' \supseteq S'$, and since the new scenario only differs from the original in that all busy processors enable two nodes, then: 1. $G' = B'$; 2. $B' = B$; 3. $G' \supseteq G$; and 4. $S' \cap G = S$. Thus, $d' = P(1-\alpha)$. We now consider the two possible situations:

$G' = G$ — In this case $G = B$, implying $spreads(p, \alpha, d) = spreads(p, \alpha, P(1-\alpha))$.

$G' \supset G$ — If p targets a processor whose *state* flag is set to WORKING, then its spread attempt fails. Thus, in this case p would not spread a node, regardless of the number of processors that enable two nodes. However, if p targets a processor whose *state* flag is set to IDLE, then its attempt has a chance to succeed. Suppose p targets some processor q whose *state* flag is set to IDLE. Then, one of the following situations occurs:

$S' = S$ In this case, the chances that p's spread attempt succeeds are the same, implying $\mathit{spreads}(p, \alpha, d) = \mathit{spreads}(p, \alpha, d') = \mathit{spreads}(p, \alpha, P(1 - \alpha))$.

$S' \supset S$ Note that, thanks to the synchronous environment we have artificially created (and assuming that any call to $UniformlyRandomNumber$ takes the same number of steps), every processor executes the CAS instruction — whose success dictates the success of the spread attempt — at the same step (line 37 of Algorithm 1). Due to our assumptions regarding the CAS instruction (see the second paragraph of Section 3) and since processors target donees uniformly at random, the greater the number of spread attempts, the greater the number of spread attempts that target q, and the smaller are the chances for p's spread attempt to succeed, implying $\mathit{spreads}(p, \alpha, d) > \mathit{spreads}(p, \alpha, d') = \mathit{spreads}(p, \alpha, P(1 - \alpha))$.

■

Lemma 4.10. *Suppose there are B bins, each of which is painted either red or green, and let B_R and B_G denote the initial number of red and green bins, respectively. Additionally, let α denote the initial ratio of red bins, meaning $\alpha = \frac{B_R}{B}$ and $B\,(1 - \alpha) = B_G$. Finally, consider that each bin stores one ball, which is painted white.*
Now, suppose there are B_R cubes and B_R pyramids. First, each pyramid is tossed, independently and uniformly at random, into the bins. Next, for each green bin that contains at least one pyramid, collect the (white) ball that is stored in the bin and paint the ball black. After inspecting all the green bins, and thus collecting and painting all the balls that were to be collected and painted, each of the B_R cubes is tossed, independently and uniformly at random, into the bins. After tossing all the cubes, count the number of cubes that landed in green bins, and, for each such cube, a red bin is painted green. Finally, toss each of the black balls (that were previously collected and painted) independently and uniformly at random, into the bins.
Let Y denote the number of bins that are still red, with at least one black ball. Then

$$\mathsf{E}[Y] \;\geq B\alpha^2\big(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\big).$$

Proof of Lemma 4.10. Let C_{Ghit} and $B_{R\mapsto G}$ be two random variables, corresponding, respectively, to the number of cubes that land in green bins and to the number of red bins that are painted green. Then, $B_{R\mapsto G} = C_{Ghit}$, and thus

$$\begin{aligned}\mathsf{E}[B_{R\mapsto G}] &= \mathsf{E}[C_{Ghit}] \\ &= B\alpha(1 - \alpha).\end{aligned}$$

Similarly to Lemma 4.7, for a bin b_i, let X_i be an indicator variable, defined as

$$X_i = \begin{cases} 1 & \text{if at least one pyramid lands in } b_i; \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the probability that at least one of the B_R pyramids lands in b_i is

$$\begin{aligned}\mathsf{P}\{X_i = 1\} &= 1 - \left(1 - \frac{1}{B}\right)^{B\alpha} \\ &\geq 1 - e^{-\alpha}.\end{aligned}$$

Let us now consider a simpler problem in which only the ball of (some) bin b_j may be collected. For a fixed (but arbitrary) bin b_i, let S_i denote a random variable, corresponding to the number of black balls landing in b_i. The ball stored in bin b_j will not land in b_i *iff* one of the following scenarios takes place:

1. The ball is not collected. The probability associated with this situation is

$$1 - \mathsf{P}\{X_j = 1\}$$

2. The ball is collected, but lands in some other bin. The probability associated with this last case is

$$\mathsf{P}\{X_j = 1\}\left(1 - \frac{1}{B}\right)$$

Thus,

$$\begin{aligned}
\mathsf{P}\{S_i = 0\} &= (1 - \mathsf{P}\{X_j = 1\}) + \mathsf{P}\{X_j = 1\}\left(1 - \frac{1}{B}\right) \\
&= 1 - \mathsf{P}\{X_j = 1\} + \left(\mathsf{P}\{X_j = 1\} - \frac{\mathsf{P}\{X_j = 1\}}{B}\right) \\
&= 1 - \frac{\mathsf{P}\{X_j = 1\}}{B} \\
&\leq 1 - \frac{(1 - e^{-\alpha})}{B}
\end{aligned}$$

Generalizing for the possible B_G balls that may be collected, it follows:

$$\begin{aligned}
\mathsf{P}\{S_i = 0\} &= \left(1 - \frac{\mathsf{P}\{X_j = 1\}}{B}\right)^{B_G} \\
&\leq \left(1 - \frac{(1 - e^{-\alpha})}{B}\right)^{B(1-\alpha)} \\
&\leq e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}.
\end{aligned}$$

Let Y_i be an indicator variable for bin b_i, such that

$$Y_i = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, $\mathsf{P}\{Y_i = 0\} = \mathsf{P}\{S_i = 0\}$, implying

$$\begin{aligned}
\mathsf{P}\{Y_i = 1\} &= 1 - \mathsf{P}\{S_i = 0\} \\
&\geq 1 - e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}.
\end{aligned}$$

Since the probability that no ball lands in bin b_i is independent from the number of red bins turning green (*i.e.* Y_i and $B_{R\mapsto G}$ are independent), then, for any m

$$\mathsf{P}\{Y_i = 0 | B_{R\mapsto G} = m\} = \mathsf{P}\{Y_i = 0\}$$

and

$$\mathsf{P}\{Y_i = 1 | B_{R\mapsto G} = m\} = \mathsf{P}\{Y_i = 1\}.$$

Suppose $B_{R\mapsto G} = m$. It follows

$$\begin{aligned}
\mathsf{E}[Y_i | B_{R\mapsto G} = m] &= 0.\mathsf{P}\{Y_i = 0 | B_{R\mapsto G} = m\} + 1.\mathsf{P}\{Y_i = 1 | B_{R\mapsto G} = m\} \\
&= \mathsf{P}\{Y_i = 1 | B_{R\mapsto G} = m\} \\
&= \mathsf{P}\{Y_i = 1\} \\
&\geq 1 - e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}.
\end{aligned}$$

Y denotes the number of bins that are still red with at least one ball:

$$Y = \sum_{i=1}^{B_R - B_{R\mapsto G}} Y_i$$

Then, by the linearity of expectation we have

$$\begin{aligned}
\mathsf{E}[Y|B_{R\mapsto G}=m] &= \mathsf{E}[Y_1+Y_2+\ldots+Y_{B_R-m}|B_{R\mapsto G}=m] \\
&= \sum_{i=1}^{B_R-m} \mathsf{E}[Y_i|B_{R\mapsto G}=m] \\
&\geq \sum_{i=1}^{B_R-m} \left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right) \\
&= (B_R-m)\left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right).
\end{aligned}$$

By the law of total expectation, it follows

$$\begin{aligned}
\mathsf{E}[Y] &= \sum_{m=0}^{B_R} \mathsf{E}[Y|B_{R\mapsto G}=m]\mathsf{P}\{B_{R\mapsto G}=m\} \\
&\geq \sum_{m=0}^{B_R} (B_R-m)\left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right)\mathsf{P}\{B_{R\mapsto G}=m\} \\
&= \left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right)\sum_{m=0}^{B_R} (B_R-m)\,\mathsf{P}\{B_{R\mapsto G}=m\} \\
&= \left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right)\left(\sum_{m=0}^{B_R} B_R\mathsf{P}\{B_{R\mapsto G}=m\} - \sum_{m=0}^{B_R} m\mathsf{P}\{B_{R\mapsto G}=m\}\right) \\
&= \left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right)\left(B_R-\mathsf{E}[B_{R\mapsto G}]\right) \\
&= \left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right)\left(B_R-B\,(1-\alpha)\,\alpha\right) \\
&= \left(B\alpha-B\alpha\,(1-\alpha)\right)\left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right) \\
&= B\alpha^2\left(1-e^{-(1-\alpha)\left(1-e^{-\alpha}\right)}\right).
\end{aligned}$$

■

We now obtain lower bounds on the total number of spreads (or donations) made to processors during the second/third phases of some scheduling iteration, assuming that all busy processors enable two nodes.

Lemma 4.11. *Consider any round $\bar{t}$ during a computation's execution under WSS, and let $B_{\bar{t}}$ be the set of processors that are busy during $\bar{t}$. If $\forall p \in B_{\bar{t}}, |E_{\bar{t}}(p)| = 2$, then $\mathsf{E}[|Spread^{+}_{\bar{t}}(B_{\bar{t}})|] \geq P\alpha_{\bar{t}}^2\left(1-e^{-(1-\alpha_{\bar{t}})\left(1-e^{-\alpha_{\bar{t}}}\right)}\right)$.*

Proof of Lemma 4.11. We prove this result by making an analogy with Lemma 4.10 where: 1. the number of bins B corresponds to the number of processors P; 2. the initial ratio of red and green bins correspond, respectively, to the ratio of idle and busy processors during the round; 3. each pyramid toss corresponds to a thief (*i.e.* an idle processor) updating some other processor's *spreading* flag; 4. each ball that is collected corresponds to a busy processor whose *spreading* flag was set to TRUE; 5. each cube toss corresponds to a steal attempt; 6. each red bin that is painted green corresponds to a processor that was idle but whose steal attempt succeeded, and thus changed its *state* flag to WORKING; and 7. each ball toss corresponds to a spread attempt.

Note that this analogy is consistent with what we want to prove:

1. Obtaining lower bounds on the probability that at least one pyramid lands in a bin corresponds to obtaining lower bounds on the probability that a processor's *spreading* flag is set to TRUE;

2. Obtaining upper bounds on the expected number of cubes landing in a bin corresponds to obtaining upper bounds on the expected number of nodes stolen from a processor. Consequently, by multiplying these bounds by the number of non-self-stable processors, we obtain upper bounds on the expected number of nodes stolen by all processors, which corresponds to the expected number of processors that were idle at the beginning of the iteration, but successfully stole a node and thus set their *spreading* flag to TRUE.

3. Obtaining lower bounds on the expected number of bins that are still red with at least one ball corresponds to obtaining lower bounds on the expected number of processors that did not successfully steal a node during the first phase, but that were targeted by a spread attempt in the following phase.

4. All steal attempts (and consequent *state* flag updates) take place during the first phase of scheduling iterations while all spread attempts take place during the second phase.

Thus, $\mathsf{E}[|Spread^{+}_{\bar{t}}\,(B_{\bar{t}})\,|] \geq P\alpha_{\bar{t}}^{2}\left(1 - e^{-(1-\alpha_{\bar{t}})\left(1-e^{-\alpha_{\bar{t}}}\right)}\right)$. ■

Lemma 4.12. $\mathsf{E}[|Spread^{+}_{\bar{t}}\,(p)\,|] \geq \frac{\alpha_{\bar{t}}^{2}}{1-\alpha_{\bar{t}}}\left(1 - e^{-(1-\alpha_{\bar{t}})\left(1-e^{-\alpha_{\bar{t}}}\right)}\right)$.

Proof of Lemma 4.12. By Lemma 4.9 it follows that $\mathsf{E}[|Spread^{+}_{\bar{t}}\,(p_{\bar{t}})\,|]$ is the smallest if all busy processors enable two nodes. By Lemma 4.11, the expected number of nodes spread during a round such that all busy processors make a spread attempt is at least $P\alpha_{\bar{t}}^{2}\left(1 - e^{-(1-\alpha_{\bar{t}})\left(1-e^{-\alpha_{\bar{t}}}\right)}\right)$. Since, as we already noted, all processors have the same chances to make a successful spread attempt, and because each spread attempt may migrate at most one node, it follows that the expected number of nodes spread by each processor that makes a spread attempt is the same. Thus, since the expected number of nodes that p spreads is the smallest if all processors make a spread attempt, then, letting $B_{\bar{t}}$ denote the set of processors that are busy during $\bar{t}$, it follows

$$\mathsf{E}[|Spread^{+}_{\bar{t}}\,(p_{\bar{t}})\,|] = \frac{\mathsf{E}[|Spread^{+}_{\bar{t}}\,(B_{\bar{t}})\,|]}{P(1-\alpha_{\bar{t}})} \geq \frac{\alpha_{\bar{t}}^{2}}{1-\alpha_{\bar{t}}}\left(1 - e^{-(1-\alpha_{\bar{t}})\left(1-e^{-\alpha_{\bar{t}}}\right)}\right).$$

■

To prove the following result, we will use two auxiliary claims.

Lemma 4.13. $\forall\alpha \in [0.8675; 1[, \quad 1 < 1 - e^{-\alpha} + \frac{\alpha^{2}}{1-\alpha}\left(1 - e^{-\left(1-e^{-\alpha}\right)(1-\alpha)}\right)$.

Claim 4.14. $\forall\alpha \in [0.8675; 1[$,

$$-e^{-\alpha} + \frac{\alpha^{2}}{1-\alpha}\left(1 - e^{-\left(1-e^{-\alpha}\right)(1-\alpha)}\right) \geq -e^{-\alpha} + \frac{\alpha^{2}}{1-\alpha}\left(1 - e^{-\left(1-e^{-0.8675}\right)(1-\alpha)}\right).$$

Proof. $\forall \alpha \in [0.8675; 1[$:

$$\begin{aligned}
e^{-0.8675} &\geq e^{-\alpha} \\
1 - e^{-0.8675} &\leq 1 - e^{-\alpha} \\
-(1 - e^{-0.8675})(1-\alpha) &\geq -(1-e^{-\alpha})(1-\alpha) \\
e^{-(1-e^{-0.8675})(1-\alpha)} &\geq e^{-(1-e^{-\alpha})(1-\alpha)} \\
1 - e^{-(1-e^{-0.8675})(1-\alpha)} &\leq 1 - e^{-(1-e^{-\alpha})(1-\alpha)} \\
\frac{\alpha^2}{1-\alpha}\big(1 - e^{-(1-e^{-0.8675})(1-\alpha)}\big) &\leq \frac{\alpha^2}{1-\alpha}\big(1 - e^{-(1-e^{-\alpha})(1-\alpha)}\big) \\
-e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\big(1 - e^{-(1-e^{-0.8675})(1-\alpha)}\big) &\leq -e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\big(1 - e^{-(1-e^{-\alpha})(1-\alpha)}\big)
\end{aligned}$$

■

Claim 4.15. Let $r = 1 - e^{-0.8675}$, and

$$s(\alpha) = e^{r(-1+\alpha)}(-2 + \alpha - r\alpha + r\alpha^2) + 2 - \alpha.$$

Then,

$$\forall \alpha \in [0.8675; 1[\,, \quad \frac{\mathrm{d}s(\alpha)}{\mathrm{d}\alpha} < 0.$$

Proof. First, note that

$$\frac{\mathrm{d}s(\alpha)}{\mathrm{d}\alpha} = -1 + e^{r(-1+\alpha)}(1 - r + 2r\alpha + r(-2 + \alpha - r\alpha + r\alpha^2)).$$

Moreover, notice that $\forall \alpha \in [0.8675; 1[\,, \quad \alpha - r\alpha + r\alpha^2 < 1$. It follows:

$$\begin{aligned}
-1 + e^{r(-1+\alpha)}(1 - r + 2r\alpha + r(-2 + \alpha - r\alpha + r\alpha^2)) &< -1 + e^{r(-1+\alpha)}(1 - r + 2r\alpha + r(-2+1)) \\
&= -1 + e^{r(-1+\alpha)}(1 - r + 2r\alpha - r) \\
&= -1 + e^{r(-1+\alpha)}(1 - 2r(1-\alpha)) \\
&< -1 + 1 - 2r(1-\alpha) \\
&= -2r(1-\alpha) \\
&< 0.
\end{aligned}$$

■

Proof of Lemma 4.13.

$$1 < 1 - e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\left(1 - e^{-\left(1-e^{-\alpha}\right)(1-\alpha)}\right)$$

iff

$$0 < -e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\left(1 - e^{-\left(1-e^{-\alpha}\right)(1-\alpha)}\right)$$

Taking into account Claim 4.14, it is *STP*

$$\forall \alpha \in [0.8675; 1[\,, \quad 0 < -e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\left(1 - e^{-\left(1-e^{-0.8675}\right)(1-\alpha)}\right).$$

Let $r = 1 - e^{-0.8675}$, and

$$u(\alpha) = -e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\left(1 - e^{-r(1-\alpha)}\right).$$

Then,

$$\frac{\mathrm{d}u(\alpha)}{\mathrm{d}\alpha} = e^{-\alpha} + \frac{\alpha}{(-1+\alpha)^2}(e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha).$$

Let

$$s(\alpha) = e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha.$$

By Claim 4.15, $s(\alpha)$ strictly decreases for $\alpha \in [0.8675; 1[$, implying

$$\begin{aligned} e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha &> e^{r(-1+1)}(-2+1-r.1+r.1^2)+2-1 \\ &= e^0(-2+1-r+r)+1 \\ &= 0, \end{aligned}$$

it follows

$$\begin{aligned} e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha &> 0 \\ \frac{\alpha}{(-1+\alpha)^2}(e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha) &> 0 \\ e^{-\alpha} + \frac{\alpha}{(-1+\alpha)^2}(e^{r(-1+\alpha)}(-2+\alpha-r\alpha+r\alpha^2)+2-\alpha) &> e^{-\alpha} > 0. \end{aligned}$$

Thus,

$$\forall \alpha \in [0.8675; 1[\,, \quad \frac{\mathrm{d}u(\alpha)}{\mathrm{d}\alpha} > 0,$$

meaning that $u(\alpha)$ strictly increases for $\alpha \in [0.8675; 1[$. To conclude this proof, note that $\forall \alpha \in [0.8675; 1[$:

$$\begin{aligned} u(\alpha) &\geq u(0.8675) \\ &= -e^{-0.8675} + \frac{0.8675^2}{1-0.8675}\left(1 - e^{-(1-e^{-0.8675})(1-0.8675)}\right) \\ &> 0. \end{aligned}$$

■

By Lemmas 4.4 and 4.5, it follows $\left|M^+_{\bar{t}}(p)\right| = \left|Stolen^+_{\bar{t}}(p)\right| + \left|Spread^+_{\bar{t}}(p)\right|$, and by Lemmas 4.8 and 4.12, it follows $\mathsf{E}[\left|M^+_{\bar{t}}(p)\right|] \geq 1 - e^{-\alpha_{\bar{t}}} + \frac{\alpha^2_{\bar{t}}}{1-\alpha_{\bar{t}}}\left(1 - e^{-(1-\alpha_{\bar{t}})}\right)$. Thus, taking into account Lemma 4.13, having $\alpha = \alpha_{\bar{t}}$, we conclude the proof of Theorem 4.1.

5 Related work

To the best of our knowledge there is no communication efficient scheduler of structured computations that overcomes WS's performance limitations nor that relies on a spreading mechanism. The work that most closely relates to this paper is [?], where GWSS, the Greedy Work Stealing and Spreading scheduler for structured computations, is presented. In that paper, the authors establish a formal framework for analyzing structured computation schedulers, define a criterion for measuring their performance, and present a methodology for studying the performance of randomized structured computation schedulers. Additionally, the authors prove that not only WS's performance is limited — which they do by showing that WS is not *algorithm short-term stable wrt* any non-empty interval —, but also that GWSS (referred to in their paper as the Work Stealing and Spreading algorithm) is *algorithm short-term stable wrt* $[0.7375, 1[$, implying that it overcomes WS's performance limitations. The algorithm presented in this paper, WSS, greatly improves upon GWSS as, in addition to overcoming WS's limitations, it maintains WS's high communication efficiency. Although the analysis of WSS is based on GWSS's, it is rendered non-trivial due to the inclusion of the *spreading* flag — used to avoid unnecessary spread attempts — which causes the number of spread attempts made during a round to be random, and thus making the analysis significantly more complex.

Except for [**?**], most theoretical work dealing with the study of online structured computation schedulers has focused on proving properties related with the full execution of computations by schedulers. Blelloch *et al.* presented the first provably time and space efficient scheduler of structured computations [7, 8]. In [9], Blumofe *et al.* present and study a Work Spreading algorithm with provably good execution performance and whose space requirements are only a logarithmic factor away from optimal. Unfortunately, the amount of communication used by their algorithm is far from optimal as processors periodically send work to each other. Later, Blumofe *et al.* presented the first WS (Work Stealing) algorithm that is optimal up to a constant factor in terms of space requirements, expected execution time, and expected communication costs [10]. Arora *et al.* extended WS for dealing with multi-programmed environments, and proved that the variant is asymptotically optimal even when executing computations under such environments [4, 5]. Many other variants of WS have been studied in the literature, each addressing a different aspect of WS [2, 3, 20], but, to the best of our knowledge, none effectively deals with the limitation of the WS algorithm for structured computations. In the context of independent task schedulers, Berenbrink *et al.* studied the performance of an independent task work stealing algorithm where each steal is allowed to take up to half of a processor's work, and proved that the scheduler is stable for a long term execution [6]. This strategy has been especially successful for structured computation schedulers that operate on distributed memory environments, where each steal attempt incurs in significant latency making it worth to transfer a larger amount of work in a single steal [2, 15, 16, 21]. Nevertheless, and despite relying on a purely receiver-initiated load balancer, the Steal-Half Work Stealing algorithm is way less communication efficient than WSS and WS as the same work can be migrated over and over again, swinging from processor to processor.

6 Conclusion and future work

The main motivation for using Work Stealing algorithms over Work Spreading ones for scheduling structured computations lies in communication costs [10, 13, 17, 23]: while in Work Stealing schedulers a processor only communicates with others when it is idle, searching for work, in Work Spreading schedulers busy processors have at least to periodically check for idle ones in order to send them work [2, 9]. Thus, in scenarios where all processors are busy Work Stealing algorithms do not use communication whilst Work Spreading ones do. In fact, Work Stealing is so communication efficient that the expected total amount of communication incurred during the execution of a computation using the WS (Work Stealing) algorithm presented in [10] is asymptotically optimal [10, 23]. Nevertheless, as proved in [**?**], its performance is limited.

In this paper, we presented and analyzed a communication efficient scheduler of structured computations that mixes Work Stealing with Work Spreading in order to overcome the limitations of WS. The WSS algorithm is a variant of GWSS [**?**] that, by only allowing at most one spread attempt for each steal attempt, guarantees that the total extra communication imposed by the spreading mechanism is amortized by the communication costs incurred by steal attempts, making it highly communication efficient. Further, our analysis shows that WSS is *algorithm short-term stable wrt* $[0.8675, 1[$, which informally means that if at least 86.75% of the processors are idle, then the expected amount of work executed by processors surpasses the maximum amount of work that can be possibly generated. Even though this result may seem rather weak, especially when taking into account that GWSS is *algorithm short-term stable wrt* $[0.7375, 1[$ (see [**?**]), note that our analysis is, as it had to be, quite pessimistic: since the whole analysis relies on studying WSS's performance on a round basis, we have to assume the worst case scenario, which, for WSS, means assuming that at the beginning of the round all processors have their *spreading* flag set to FALSE. As a matter of fact, we believe that in more realistic scenarios, if the ratio of idle processors were high then most processors would have their *spreading* flags set to TRUE due to the high chances of being targeted by other processors, and so the behavior of WSS would approximate GWSS's.

Although WSS enjoys the benefits of both WS and GWSS, we still do not know if it is possible to come up with a practical implementation of the scheduler for asynchronous, multi-programmed environments that is competitive with state-of-the-art implementations of WS. As such, it would be very interesting to see how the spreading mechanism of WSS could be implemented in practice without significantly increasing the overheads of task creation, and, especially, without introducing a new source of expensive synchronization. In addition to a practical implementation, it would also be interesting to study the performance of a variant of WSS where processors were allowed to keep their *spreading* flag set to TRUE after performing a successful spread attempt, rather than immediately resetting it back to FALSE, and thus permitting *"lucky"* processors to make a series of successful spreads instead of a single one.

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

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

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

[6] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

[7] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *SPAA*, pages 1–12, 1995.

[8] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[9] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

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

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

[12] Robert David Blumofe. *Executing multithreaded programs efficiently.* PhD thesis, Massachusetts Institute of Technology, 1995.

[13] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture, FPCA 1981, Wentworth, New Hampshire, USA, October 1981*, pages 187–194, 1981.

[14] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 536–545, 2008.

[15] 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.

[16] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 280–289, 2002.

[17] Robert H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LISP and Functional Programming*, pages 9–17, 1984.

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

[19] 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.

[20] 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.

[21] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*, pages 217–229, 2010.

[22] 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.

[23] 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 The lock-free deque semantics

In this section, we present the specification of the relaxed semantics associated with the lock free deque's implementation as given in [5]. The deque implements three methods:

1. *pushBottom* – adds an item to the bottom of the deque and does not return.

2. *popBottom* – returns the bottom-most item from the deque, or EMPTY, if there is no node.

3. *popTop* – attempts to return the topmost item from the deque, or EMPTY, if there is no node. If the attempt succeeds, a node is returned. Otherwise, the special value ABORT is returned.

The implementation is said to be *constant-time iff* any invocation to each of the three methods takes at most a constant number of steps to return, implying the sequence of instructions composing the invocation has constant length.

An invocation to one of the deque's methods is defined by a 4-tuple establishing: 1. the method invoked; 2. the invocation's beginning time; 3. the invocation's completion time; and 4. the return value, if it exists.

A set of invocations meets the *relaxed semantics iff* there is a set of *linearization times* for the corresponding non-aborting invocations for which: 1. every non-aborting invocation's linearization time lies within the initiation and completion times of the respective invocation; 2. no two linearization times coincide; 3. the return values for each non-aborting invocation are consistent with a serial execution of the methods in the order given by the linearization times of the non-aborting invocations; and 4. for each aborted *popTop* invocation x to a deque d, there is another invocation removing the topmost item from d whose linearization time falls between the beginning and completion times of x's invocation.

A set of invocations is said to be *good iff pushBottom* and *popBottom* are never invoked concurrently. The deque implementation presented in [5] has been proven to satisfy the relaxed semantics on any good set of invocations [11]. Note that any set of invocations made during the execution of a computation scheduled by WSS is good, as the *pushBottom* and *popBotom* methods are exclusively invoked by the (unique) owner of the deque. Thus, throughout the paper we simply assume that the relaxed semantics are met.

We assume a deque implementation equivalent to the one given in [5], which was proved to be a correct and constant-time implementation meeting the afore mentioned description. We assume that the implementation of the deque is constant time, meaning that any invocation to each of these methods takes at most a constant number of steps to return.