On the analysis of scheduling algorithms 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. Although it delivers provably good performances, the effectiveness of its underlying load balancing strategy is known to be limited for some classes of computations. Many studies have addressed this limitation from a purely load balancing perspective, considering that computations are composed by a set of independent tasks and then analyzing the expected amount of work attached to each processor as the execution progresses. To that end, these studies make assumptions on work generation that, although are realistic from a queuing theory perspective, do not match the reality of structured parallel computations.

In this paper, we introduce a formal framework for studying the performance of structured computation schedulers, define a criterion that is appropriate for measuring their performance, and present a methodology to analyze the performance of randomized schedulers. We demonstrate the convenience of this methodology by using it to prove that the performance of Work Stealing is limited, and to analyze the performance of a Work Stealing and Spreading algorithm, which overcomes Work Stealing's limitation.

1 Introduction

The main goal of a structured computation's scheduler is to guarantee the fast completion of the execution of arbitrary structured computations. For some time now, Work Stealing is one of the most popular algorithms for scheduling structured computations [7, 8, 12]. 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 [8, 12], the expected execution time of any computation using WS is asymptotically optimal. Yet, WS's performance is known to be limited for the execution of computations that exhibit unbalanced parallelism (*e.g.* depth first computations where only a few threads actually generate work) [2, 10, 13, 14]. For coping with this limitation, numerous studies have been resorting to the use of steal-half deques [2, 14, 17, 27] which allow thieves to take up to half of the work of their victims. The adoption of steal-half strategies by real-life schedulers has been mostly justified by their importance 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. On the other hand, the steal-half strategy has been formally proved, from a queuing theory perspective, to be an effective load balancing method for schedulers of independent tasks [10]. However, while this strategy may be ideal for independent task scheduling from a queuing theory perspective — where tasks are assumed to arrive at a system according to some probability distribution, and work transfers are assumed to take constant time regardless of the amount of tasks transferred — it remains unknown whether it is suitable for structured computation scheduling — where work generation only depends on the structure of a computation, and where the time for a processor to transfer work from another processor is proportional to the amount of work transferred —, and so the problem of how to cope with the limitation of WS remains open. Even more importantly, while there are well established methods for analyzing the performance of the load balancers of independent task schedulers — usually based on the analysis of Markov chains — and a well-defined goal — which is usually to assure that the system's load does not grow unboundedly over time —, to the best of our knowledge there are no well-defined methods suitable for the performance analysis of the load balancers of online structured computation schedulers, nor even well-defined goals.

To this extent, the contributions of this paper are:

- A formal framework for studying the performance of structured computation schedulers (Section 2). One of the key features of this framework is that it can be used to model most, if not all, practical scheduling algorithms.
- The definition of *algorithm short-term stability* (Section 2.1), which is an appropriate criterion for measuring the performance of online structured computation schedulers.
- A methodology that allows to effectively study the performance of randomized computation schedulers (Section 3). We demonstrate its convenience by: 1. using it to prove that the performance of WS is indeed limited (Section 3.2); and 2. presenting a variant of the WS algorithm where processors attempt to spread work as it is generated, and then using our methodology to show that the algorithm overcomes the identified limitations of WS (Section 4).

2 The formal framework

Like in much previous work [1, 6, 8, 12, 26, 29], we model a 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.

We consider that processors operate on discrete time steps, each executing one instruction — that may or may not correspond to a computation node — per time step. The execution of a computation 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.* $Procs$ is composed by at least two processors), and that all processors operate synchronously in time steps. Therefore, a computation's execution can be partitioned into discrete time steps, 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 and $i+1$ is the step succeeding i.

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

As one may note, a node can only be **ready** if all the ordering constraints wrt (with respect to) the node are satisfied. For example, at the first step of a computation's execution every node (except for the root) is **not ready**. To ensure the correct execution of a computation, 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 $Executed_i$. Since only the nodes that are **ready** can become **executed**, $Executed_i$ can alternatively be defined as $Executed_i = \left(\bigcup_{j\in\{1,\ldots,i-1\}} R_j\right) - R_i$.

For each step i, partition R_i into P sets (one per processor), and refer to $R_i(p)$ — p's partition of R_i — as the set of nodes that are *attached* to p at step i. Say that a node was *enabled* at step i if it was **not ready** at step i but is **ready** at step $i+1$, and, similarly, that a node was *executed* at step i if it was **ready** at step i but is **executed** at step $i+1$. In addition, say that a node μ is *migrated* if $\mu \in R_i\left(p\right)$ and $\mu \in R_{i+1}\left(q\right)$, where $p \neq q$, for $p, q \in Procs$.

Definition 2.2. For each step i and processor p, define the set of nodes *enabled* by p as $E_i(p) = R_{i+1}(p) - R_i$ and *executed* (or *computed*) by p as $C_i(p) = R_i(p) - R_{i+1}$. Moreover, 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 $M_i^-(S) = \bigcup_{p\in S} M_i^-(p)$. Additionally, 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 C time steps (for some constant $C \geq 1$) such that 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 C denote the length of rounds and $\overline{t}\left[i\right]$ denote the *i-th* step of a round $\overline{t}$ (for $i \in \{0, \ldots, C-1\}$). As we will see, the length of rounds depends on the scheduling algorithm.

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 $E_{\overline{t}}(p) = \bigcup_{i\in\{\overline{t}[0],\cdots,\overline{t}[C-1]\}} E_i(p)$, *executed* by p during $\overline{t}$ as $C_{\overline{t}}(p) = \bigcup_{i\in\{\overline{t}[0],\cdots,\overline{t}[C-1]\}} C_i(p)$, *migrated* from p to all other processors during $\overline{t}$ as $M_{\overline{t}}^+(p) = \bigcup_{i\in\{\overline{t}[0],\cdots,\overline{t}[C-1]\}} M_i^+(p)$ and *migrated* from all other processors to p during $\overline{t}$ as $M_{\overline{t}}^-(p) = \bigcup_{i\in\{\overline{t}[0],\cdots,\overline{t}[C-1]\}} M_i^-(p)$.

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 $M_{\overline{t}}^-(S) = \bigcup_{p\in S} M_{\overline{t}}^-(p)$. Additionally, define

$E_{\bar{t}} = E_{\bar{t}}(Procs)$, $C_{\bar{t}} = C_{\bar{t}}(S)$, $M^+_{\bar{t}} = M^+_{\bar{t}}(Procs)$, and $M^-_{\bar{t}} = M^-_{\bar{t}}(Procs)$.

The proof of the following result can be found in the Appendix (Section A.1).

Lemma 2.5. *For any round $\bar{t}$ and $p \in Procs$, $M^-_{\bar{t}}(p) \subseteq M^+_{\bar{t}}(Procs - \{p\})$.*

As one may note, Definition 2.3 implies that for any processor p and round $\bar{t}$, $|C_{\bar{t}}(p)| \leq 1$ and $M^+_{\bar{t}}(p) \cap M^+_{\bar{t}}(Procs - \{p\}) = \emptyset$ (*i.e.* no two processors migrate the same node during the same round). By Lemma 2.5, it then follows $M^+_{\bar{t}}(p) \cap M^-_{\bar{t}}(p) = \emptyset$.

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

Requirement 2.6. For any round $\bar{t}$ and $p \in Procs$, we must have $R_{\bar{t}}(p) \supseteq C_{\bar{t}}(p)$.

The next lemma is essential for the rest of our analysis, as it shows the connection between the set of nodes that are attached to each processor p at some round $\bar{t}$, and the set of nodes that are attached to p at round $\overline{t+1}$. The proof of this result can be found in the Appendix (Section A.2).

Lemma 2.7 (Round Progression Lemma)**.** *For any round $\bar{t}$ and processor $p \in Procs$,*

$$R_{\overline{t+1}}(p) = \left(E_{\bar{t}}(p) \cup R_{\bar{t}}(p) \cup M^-_{\bar{t}}(p)\right) - \left(C_{\bar{t}}(p) \cup M^+_{\bar{t}}(p)\right).$$

2.1 A criterion to measure the performance of computation schedulers

We now move to present the criterion that will be used to measure the performance of structured computation schedulers.

Definition 2.8. Say that a processor $p \in Procs$ is *idle* during a round $\bar{t}$ if $C_{\bar{t}}(p) = \emptyset$, and, otherwise, say that p is *busy*. Moreover, denote the number of idle processors during a round $\bar{t}$ by $P^{idle}_{\bar{t}}$, and define $\alpha_{\bar{t}}$ as the ratio of idle processors, $\alpha_{\bar{t}} = P^{idle}_{\bar{t}}/P$.

Now, we introduce the notion of short-term stability. Intuitively, a set of processors S is short-term stable for some round $\bar{t}$ if the number of nodes attached to the processors in S, that are not executed, is expected to **monotonically** decrease from round $\bar{t}$ to round $\overline{t+1}$.

Definition 2.9 (Short-term stability)**.** A set of processors $S \in \mathcal{P}(Procs)$ is short-term stable for some round $\bar{t}$ during a computation's execution if $\mathrm{E}[|R_{\overline{t+1}}(S) - C_{\overline{t+1}}(S)|] \leq |R_{\bar{t}}(S) - C_{\bar{t}}(S)|$.

Ideally, we would want to ensure short-term stability for all rounds and wrt all processors. However, since a processor can enable two nodes during one round, a scheduler may only be able to guarantee short-term stability wrt all processors if at least half of them are idle during a round.

The proof of the next lemma can be found in the Appendix (Section A.3).

Lemma 2.10. *For any round $\bar{t}$ and $S \in \mathcal{P}(Procs)$, if $\forall p \in S$, $E\left[\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right|\right] < |R_{\bar{t}}(p) - C_{\bar{t}}(p)|$ then $E\left[\left|R_{\overline{t+1}}(S) - C_{\overline{t+1}}(S)\right|\right] < |R_{\bar{t}}(S) - C_{\bar{t}}(S)|$.*

For each round $\bar{t}$, we classify processors according to whether they execute all their attached nodes during $\bar{t}$ or not. If a processor p executes all its attached nodes during round $\bar{t}$ (*i.e.* if $R_{\bar{t}}(p) = C_{\bar{t}}(p)$), p is *self-stable* at round $\bar{t}$. Otherwise, p is *non-self-stable* at round $\bar{t}$.

Definition 2.11. Define the set of *self-stable* and *non-self-stable* processors at some round $\bar{t}$ as $S_{\bar{t}} = \{p \in Procs \mid R_{\bar{t}}(p) = C_{\bar{t}}(p)\}$ and $U_{\bar{t}} = Procs - S_{\bar{t}}$, respectively.

We now finally present our criterion for measuring computation schedulers' performance.

Definition 2.12 (Algorithm short-term stability)**.** A scheduling algorithm is *algorithm short-term stable* with respect to an interval $I \subseteq]0;1[$, iff (if and only if) for any round $\bar{t}$,

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

where C denotes the length of the rounds.

Informally, the main idea is that if the ratio of idle processors at some round $\overline{t}$ is sufficiently high, then the amount of work attached to non-self-stable processors is expected to decrease, and, at the same time, the amount work attached to self-stable processors does not grow unboundedly. Contrarily to *short-term stability*, *algorithm short-term stability* requires that the expected number of nodes attached to processors of $U_{\overline{t}}$, that are not executed, **strictly** decreases from a round to the next. By limiting the number of nodes that can become attached to a self-stable processor during a round, we disallow scheduling algorithms to keep ping-ponging work between non-self-stable and self-stable processors throughout the execution. The insight for bounding the number of nodes to the length of rounds is that we are enforcing processors to have to accept each node they are given.

The following results are base for the analysis of schedulers, as they relate, 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 a result that is closely related with *algorithm short-term stability* (the corresponding proof can be found in the Appendix, Section A.4).

Lemma 2.13 (Connecting Lemma)**.** *For any round $\overline{t}$ and processor $p \in Procs$, $|E_{\overline{t}}(p)| < \left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\overline{t}}(p)\right|$ iff $\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right| < |R_{\overline{t}}(p) - C_{\overline{t}}(p)| + \left|M^{-}_{\overline{t}}(p)\right|$.*

Corollary 2.14. *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 $\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right| < |R_{\overline{t}}(p) - C_{\overline{t}}(p)|$.*

3 A method to analyze randomized schedulers

In order to analyze the performance of randomized scheduling algorithms, we introduce a few additional definitions and make some assumptions that are necessary to permit ordering the actions that processors take during the execution of computations, and, in particular, during each round. The reason for the need of ordering processors' actions will become apparent as we use it to analyze the WS algorithm. To aid the reader, as we present the extra definitions and assumptions that our methodology requires, we use a WS algorithm (depicted in Algorithm 1) to instantiate them and explain their meaning. The WS algorithm we analyze is a synchronous but behaviorally equivalent variant of the original non-blocking algorithm given in [8]. Thus, each processor owns a 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[1]. In addition to the deque, each processor has a variable *assigned* that stores the node that it will execute next, if any.

3.1 The methodology

First of all, we require that the scheduling algorithm to be analyzed must be defined by a cycle such that: 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 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. 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*.

[1] For a more careful description of the lock-free deque semantics, originally defined in [8], please refer to Section B.

As it can be observed in Algorithm 1, the definition of the WS algorithm naturally fits into scheduling loops (corresponding to lines 2 to 19): 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 (line 23); 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.

Algorithm 1 Synchronous WS — Part 1.

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

Algorithm 2 Synchronous WS — Part 2.

```
21: procedure WORKMIGRATION
22:   $victim$ ← UniformlyRandomProcessor()
23:   $assigned$ ← $victim.deque$.popTop()
24:   synch($max\_phase_I\_length$, $\iota()$)
25: end procedure

26: function VALIDNODE($node$)
27:   return $node \neq$ EMPTY
28:       and $node \neq$ ABORT
29:       and $node \neq$ NONE
30: end function
```

To order the actions that processors take during scheduling iterations, each iteration can be partitioned into a sequence of *phases*. In particular, for WS, each iteration is partitioned into two phases:

Phase I If a processor has a valid assigned node, it executes the node. Otherwise, it makes a steal attempt, and if the attempt succeeds the stolen node becomes the processor's new assigned node.

Phase II If a processor made a steal attempt in the previous phase, it takes no action during this phase. Otherwise, the processor executed a node in the previous stage, which enabled either 0, 1 or 2 nodes. If no node was enabled, the processor invokes *popBottom* to fetch the bottommost node from its deque, if such node exists. If at least one node was enabled, one of the enabled nodes becomes the processor's new assigned node, whist the other node, if any, is pushed by the processor into the bottom of its own deque, via the *pushBottom* method.

At this point, we have already ordered the actions that each processor takes during the execution of every iteration. However, this ordering by itself does not meet our needs, as we have to guarantee that all processors start the execution of each phase of every scheduling iteration at the same time. Our first step towards that goal 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 3.1. $\forall p \in Procs, \quad \chi(p, 1) = 0.$

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, until the procedure's invocation. Given these parameters, *synch* adds a sequence of *maxPhaseLength* − *currentPhaseLength* no-op instructions, guaranteeing that the number of steps taken from the beginning of each phase's execution until the end of its call is the same for all processors. To use *synch*, we rely on the purely theoretical procedure ι to obtain the value of $currentPhaseLength$.

Lastly, we partition a computation's execution even further, by partitioning all rounds **equally** into sequences of *stages*. To formalize this idea, define a *stage partition* $\ddot{s} \in \mathbb{N} \times \mathbb{N}$, as $\ddot{s} = (base, offset)$, with $offset > 0$, where $base$ and $offset$ are, respectively, the starting step and length of the stage defined by $\ddot{s}$ within each round. Refer to the i-th stage of a round $\bar{t}$ as $\bar{t}\,\langle i \rangle$.

Definition 3.2. Let C be the length of the rounds. Say that a set $\ddot{S}$ is a set of *stage partitions* if $C = \sum_{\ddot{s}\in\ddot{S}} \pi_2(\ddot{s})$ and $\forall \ddot{s} \in \ddot{S}\left([\exists \ddot{r} \in \ddot{S} :\ \ \pi_1(\ddot{s}) + \pi_2(\ddot{s}) = \pi_1(\ddot{r})] \vee [\pi_1(\ddot{s}) + \pi_2(\ddot{s}) = C]\right)$.

Remark 3.3. To analyze a scheduler's performance using our methodology 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.

Justification. By using the *synch* and ι procedures, one can guarantee that any scheduling algorithm, that may be defined by a scheduling loop, can be modified so that processors are kept synchronized throughout any computation's execution, having that all processors begin the execution of the i-th phase of the n-th scheduling iteration at the exact same step. With this, the length of each round can be set to $\sum_{i\in Phases} length_i$, where $Phases$ denotes the set of phases that compose a scheduling iteration and $length_i$ denotes the length of the i-th phase[2]. Note that, since all processors execute each scheduling iteration synchronized, the definition of scheduling loop ensures us that the requirements of the definition of round are satisfied: 1. each round has constant length; 2. a computation's execution can be partitioned into a sequence of equal-length rounds; 3. during each round no processor executes more than a single node; and 4. no node is migrated more than once during a round. Then, it only remains to partition each round into a sequence of stages, having one stage per phase, and ensuring that the execution of the i-th phase of a scheduling iteration coincides with the i-th stage of the corresponding round. ■

The synchronous WS scheduler is depicted in Algorithm 1, where $max\_phase_{I}\_length$ and $max\_phase_{II}\_length$ are two constants that correspond to the lengths of the longest sequences of instructions composing the first and second phases of WS, respectively. Thus, by Remark 3.3, we can set the length of WS's rounds to $max\_phase_{I}\_length + max\_phase_{II}\_length$, and partition each such round into two stages whose length matches the maximum length of the corresponding phase. To proceed to analysis of WS's performance, it only remains to show that WS satisfies Requirement 2.6. For WS, say that a node μ is attached to a processor p if one of the following conditions holds: 1. μ is p's currently assigned node; 2. μ is stored in p's deque; or 3. μ is stored in $enabled\,[0]$ or $enabled\,[1]$. 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, each 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.

3.2 Work Stealing's performance

Before proving that the performance of WS is limited, we have to make an additional definition.

Definition 3.4. 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 $Stolen_{\bar{t}}^+\,(p) = \bigcup_{i\in\left\{\bar{t}[0],\ldots,\bar{t}[C-1]\right\}} Stolen_i^+\,(p)$, and the set of nodes stolen by

[2] Note that, by including the call to the *synch* procedure at the end of each phase, we ensure that the i-th phase of every scheduling iteration has the same length.

p as $Stolen^-_{\bar{t}}\,(p) = \bigcup_{i \in \left\{\bar{t}[0],\dots,\bar{t}[C-1]\right\}} Stolen^-_i\,(p)$.

The proof of the following result can be found in the Appendix (Section C.1).

Lemma 3.5. *For any round $\bar{t}$, and $p \in U_{\bar{t}}$ we have $1 - e^{-\alpha_{\bar{t}}} \leq \mathrm{E}[|M^+_{\bar{t}}(p)|] \leq \alpha_{\bar{t}}$.*

The proof of the following result can be found in Appendix (Section C.2).

Lemma 3.6. *Consider some $p \in Procs$ and some round $\bar{t}$ during the execution of a computation by WS. If $p \in U_{\bar{t}}$ then p's deque is non-empty and $M^-_{\bar{t}}\,(p) = \emptyset$. If $p \in S_{\bar{t}}$ then $\left|R_{\overline{t+1}}\,(p)\right| \leq 2$.*

Taking into account Lemma 3.6 and the definition of round (Definition 2.3), it follows $\forall p \in S_{\bar{t}}, \left|R_{\overline{t+1}}\,(p)\right| \leq 2 \leq C{+}1$ — recall that the length of rounds, C, is at least 1. Thus, if we were to show that WS is *algorithm short-term stable* wrt some interval $I \subseteq]0;1[$, then, by Corollary 2.14, we would only have to prove that for any round $\bar{t}$ such that $\alpha_{\bar{t}} \in I$, we had $|E_{\bar{t}}\,(p)| < \mathrm{E}\left[\left|C_{\overline{t+1}}\,(p)\right| + \left|M^+_{\bar{t}}\,(p)\right|\right]$. Unfortunately, as we now prove, there is no non-empty interval I wrt which WS is *algorithm short-term stable* (a full proof of Theorem 3.7 is presented in the Appendix, in Section C.3).

Theorem 3.7. *There is no non-empty interval $I \subseteq]0;1[$ such that WS (as defined in Algorithm 1) is algorithm short-term stable wrt I.*

Proof Sketch. To prove this result, we simply describe a possible round $\bar{t}$ for which it is not possible to guarantee even the short-term stability of $U_{\bar{t}}$. Consider that $U_{\bar{t}}$ is composed by a single processor p, and that p enables two nodes during round $\bar{t}$. Then, taking into account Lemma 3.5, we conclude that WS is not stable wrt any non-empty interval $I \subseteq]0;1[$. ■

4 Analyzing Work Stealing and Spreading

The Work Stealing and Spreading scheduler — or simply WSS — (depicted in Algorithm 3), is a variant of WS where processors load balance not only by stealing work, but also by spreading it. As in WS, each processor owns a lock-free deque (obeying the semantics defined in Section B) and a variable *assigned* that stores the node that it will execute next, if any. To implement the spreading mechanism each processor additionally owns a *state* flag and a *donation* cell. 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) — and use the *donation* cell to store nodes that they want to spread. In WSS 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.

Contrarily to WS, we partition each scheduling iteration of WSS into three phases. Phase I of WSS is very similar to the WS's counterpart, only differing because in WSS processors keep updating their *state* flags to reflect their current state. Phases II and III of WSS are as follows:

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. If two nodes were enabled, then, after having a new node assigned, p attempts to spread the node it did not assign.

Phase III If a processor p executed a node in phase I but no node was enabled, p invokes *popBottom* to fetch the bottommost 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 the donation attempt it made during phase II failed. In such scenario, p pushes the node it failed to donate into the bottom of its own deque, via the *pushBottom* method. Finally, if the processor made an unsuccessful steal attempt during the first phase, it polls its *state* flag to check for incoming donations. If there is a donation, p transfers the node from the donor's *donation* cell and updates its *state* flag accordingly.

Algorithm 3 WSS — Part 1.

```
procedure SCHEDULER( )
  while not finished ($computation$) do
    if ValidNode ($self.assigned$) then
      $enabled \leftarrow$ execute ($self.assigned$)
      $assigned \leftarrow$ NONE
      synch($max\_phase_I\_length$, $\iota()$)
      if length ($enabled$) $> 0$ then
        $self.assigned \leftarrow enabled\,[0]$
        if length ($enabled$) $= 2$ then
          $self$.handleExtraNode ($enabled\,[1]$)
        else
          synch($max\_phase_{II}\_length$, $\iota()$)
        end if
      else
        synch($max\_phase_{II}\_length$, $\iota()$)
        $self.assigned \leftarrow self.deque$.popBottom ()
        if not ValidNode ($self.assigned$) then
          $self.state \leftarrow$ IDLE
        end if
      end if
    else
      $self$.loadBalance ()
    end if
    synch($max\_phase_{III}\_length$, $\iota()$)
  end while
end procedure
```

Algorithm 4 WSS — Part 2.

```
procedure HANDLEEXTRANODE($\mu$)
  $self.donation \leftarrow \mu$
  $donee \leftarrow$ UniformlyRandomProcessor()
  $result \leftarrow$ CAS ($donee.state$, IDLE, $self.id$)
  synch($max\_phase_{II}\_length$, $\iota()$)
  if $result \neq$ SUCCESS then
    $self.donation \leftarrow$ NONE
    $self.deque$.pushBottom ($\mu$)
  end if
end procedure

procedure LOADBALANCE( )
  $victim \leftarrow$ UniformlyRandomProcessor()
  $self.assigned \leftarrow victim.deque$.popTop ()
  if ValidNode ($self.assigned$) then
    $self.state \leftarrow$ WORKING
  end if
  synch($max\_phase_I\_length$, $\iota()$)
  synch($max\_phase_{II}\_length$, $\iota()$)
  if $self.state \neq$ IDLE and $self.state \neq$ WORKING then
    $donor \leftarrow processor\,[self.state]$
    $self.assigned \leftarrow donor.donation$
    $self.state \leftarrow$ WORKING
  end if
end procedure
```

Definition 4.1. 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 $Spread_i^-(p)$. Moreover, for some round $\bar{t}$, define the set of nodes spread during $\bar{t}$ by p as $Spread_{\bar{t}}^+(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[C-1]\}} Spread_i^+(p)$, and the set of nodes spread to p as $Spread_{\bar{t}}^-(p) = \bigcup_{i \in \{\bar{t}[0],\ldots,\bar{t}[C-1]\}} Spread_i^-(p)$.

Taking into account the following claim and Remark 3.3, we can begin WSS's analysis. The proof of the claim can be found in the Appendix (Section D.1).

Claim 4.2. The WSS algorithm can be defined using a scheduling loop and meets Requirement 2.6.

As one might deduce, the following theorem implies that WSS overcomes the limitations of WS. A full proof of the result can be found in the Appendix (Section D.2).

Theorem 4.3. *WSS (as defined in Algorithm 3) is algorithm short-term stable wrt* $[0,7375;1[$.

Proof Sketch. Part of the proof of this theorem follows from a result that is equivalent to Lemma 3.6, but concerning WSS. The rest of the proof uses Corollary 2.14, and is by cases on how many nodes a processor $p \in U_{\bar{t}}$ might enable during any round $\bar{t}$, and the most interesting case is when p enables two nodes. Noting that p's deque is not empty, similarly to WS we obtain lower bounds on $\mathrm{E}[|Stolen_{\bar{t}}^+(p)|]$. After that, we obtain lower bounds on $\mathrm{E}[|Spread_{\bar{t}}^+(p)|]$. To conclude the proof we show that $\forall \alpha \in [0,7375;1[$, it follows $|E_{\bar{t}}(p)| < |C_{\overline{t+1}}(p)| + \mathrm{E}[|Stolen_{\bar{t}}^+(p)|] + \mathrm{E}[|Spread_{\bar{t}}^+(p)|]$. ■

5 Related work

To the best of our knowledge, there is no work that analyzes the performance of online structured computation schedulers, on a round basis, depending solely on the ratio of idle processors.

Most theoretical work dealing with the study of online structured computation schedulers, has focused on proving properties related with the (complete) execution of computations by WS and variants. Blumofe *et al.* proved that WS is optimal up to a constant factor in terms of space requirements, expected execution time, and expected communication costs [12]. Arora *et al.* showed that WS is optimal even for multiprogrammed environments [7, 8]. Agrawal *et al.* introduced a variant of WS that avoids unnecessary load balancing cycles in order to achieve higher efficiency [5, 6]. The authors proved that WS is capable of maintaining nearly optimal bounds, while reducing the number of cycles during which processors are not making progress on a computation's execution (corresponding to load balancing cycles), down to a constant factor away from the computation's total amount of work. Regarding data locality, Acar *et al.* obtained both lower and upper bounds on the number of cache misses using WS [1]. More recent research has been focusing on reducing the synchronization overheads of WS [2], mainly by eliminating synchronization for local deque operations (*i.e.* eliminating the need for synchronization when processors work locally on their own deque). Even more recently, Muller *et al.* studied the performance of WS for computations that include latency operations (such as receiving input from a user), obtaining promising results [26]. On the other hand, most practical work that deals with the scheduling of structured computations has focused either on the improvement of current WS implementations — increasing data locality [1, 16, 27, 28], reducing synchronization overheads [2, 18, 22, 25, 30], etc —, or on the development of libraries and languages implementing WS on both shared memory environments [3, 11, 15, 19, 26] and distributed settings [13, 14, 20, 27].

While, for the execution of structured computations, work generation depends on what has already been executed, for independent task scheduling, work generation (or, more correctly, task arrival) is assumed to be independent from what tasks processors already executed [4, 9, 10, 21, 23, 24]. In fact, much of the work in this area consists on studying the effectiveness of different strategies (that rely on randomness) for placing n balls (each representing a task) into n bins (each representing a processor) [4, 9, 21, 24], being that a strategy's effectiveness is measured according to the number of balls that the fullest bin is expected to have: the lower this number is, the more effective the strategy is. Of course, this type of models, despite being suitable for modelling independent task schedulers, are far from being apt to model structured computation schedulers (for example, note that in the execution of a structured computation, work is generated per processor). Within the area of independent task scheduling, perhaps the work most closely related to ours is on the performance analysis of online independent task schedulers [10, 23]. Yet, to the best of our knowledge, all the analyzes made to these schedulers rely upon the assumption that tasks arrive to the system according to some random distribution (typically Poisson's distribution). For instance, Mitzenmacher proposed a simple but powerful scheme to analyze **independent task** work stealing schedulers, that uses differential equations [23]. This scheme allows to study not only the most basic work stealing schedulers (of independent tasks), but also more complex variants (*e.g.* allowing processors to repeat a steal attempt when the previous attempt aborted). Nevertheless, the proposed scheme relies on the assumption that work is generated according to some random distribution, and so it is not suitable for modelling the behavior of structured computation schedulers. Berenbrink *et al.* study the performance of independent task work stealing schedulers, modelling the system as a Markov chain, whose states denote the number of tasks attached to each processor of the system [10]. The authors proved that the work stealing scheduler for independent tasks, where each steal is allowed to take up to half of a processor's work, is stable for a long term execution. Unfortunately, their analysis also relies on the assumption that tasks arrive at the system according to a random distribution, and so it is not apt to model the performance of structured computation schedulers. In addition, the authors assume that the number of tasks generated at each round is at most the number of processors, which, taking into account our conventions regarding the structure of computations, is not realistic for modelling schedulers of structured computations.

Although it may not be entirely straightforward, it is possible to use our methodology to model the steal-half work stealing algorithm. To do so, each steal would have to be divided into a sequence of scheduling iterations, such that during each iteration the thief transferred a node from its victim. However, transferring half of a processor's work may take some time, which not only implies that the thief will have to wait until it can begin executing what it stole, but it also means that either concurrent steal attempts to the same deque are delayed (to avoid duplicate steals), or thieves have to first transfer the work they intend to steal from their victims and only then attempt to commit the steal. Regarding the latter option, note that if a thief is transferring work from one of the only processors that is generating work, then the steal attempt is likely to fail. Moreover, since during each round a processor can enable two nodes, then, it would still be possible that the processor whose deque was being stolen generated a large amount of work.

6 Conclusion

We introduced a formal framework for the performance analysis of structured computation schedulers, and defined an appropriate criterion for measuring the performance of online scheduling algorithms: *algorithm short-term stability.* Moreover, we introduced a simple and powerful method that allows to analyze the performance of these schedulers, and have demonstrated its convenience by using it with two different ends: 1. proving that the performance of WS is indeed limited; and 2. analyzing the performance of WSS. Although WSS is a purely theoretical algorithm, its analysis gave us insight on how to possibly overcome the limitation of WS. Nevertheless, the greedy spreading strategy of the algorithm has a severe limitation that makes us question its practical value: even if every processor is busy, whenever a processor generates work it makes a spread attempt. This not only makes processors incur in unnecessary overheads (that, for modern computer architectures, are unduly large) but even more importantly, it entails a serious drawback concerning the communication costs of the algorithm. Consequently, it is still an open problem to come up with a practical algorithm that overcomes WS's limitation while maintaining its asymptotically optimal expected execution time and communication costs, and its low space requirements.

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] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Eilstrup Rasmussen. Parallel randomized load balancing. *Random Struct. Algorithms*, 13(2):159–188, 1998.

[5] 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.

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

[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] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100, 2008.

[16] Yi Guo, Jisheng Zhao, Vincent Cavé, and Vivek Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12, 2010.

[17] 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.

[18] 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.

[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] 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.

[21] Reinhard Lüling and Burkhard Monien. A dynamic distributed load balancing algorithm with provable good performance. In *SPAA*, pages 164–172, 1993.

[22] 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.

[23] Michael Mitzenmacher. Analyses of load stealing models based on differential equations. In *SPAA*, pages 212–221, 1998.

[24] Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 799–808, 2002.

[25] 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.

[26] 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.

[27] 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.

[28] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Inf. Process. Lett.*, 116(2):100–106, 2016.

[29] 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.

[30] 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.

A Full proofs for the results obtained in Section 2

A.1 Full proof for Lemma 2.5

Claim A.1. For any step i and processor p, $M_i^-(p) \subseteq M_i^+(Procs - \{p\})$.

Proof. By Definition 2.2

$$\begin{aligned}
M_i^-(p) &= R_{i+1}(p) \cap (R_i - R_i(p)) \\
&= \left(R_{i+1} - \left[\bigcup_{q \in Procs - \{p\}} R_i\left(q\right) \right] \right) \cap \left[\bigcup_{q \in Procs - \{p\}} R_i\left(q\right) \right] \\
&\subseteq R_{i+1} \cap \left[\bigcup_{q \in Procs - \{p\}} R_i\left(q\right) \right] \\
&\subseteq \left[\bigcup_{q \in Procs - \{p\}} R_i\left(q\right) \cap R_{i+1} \cap \overline{R_{i+1}\left(q\right)} \right] \\
&= M_i^+\left(Procs - \{p\}\right)
\end{aligned}$$

■

Proof of Lemma 2.5. Claim A.1 implies that for any step $i \in \{\bar{t}[0], \cdots, \bar{t}[C-1]\}$, we have $M_i^-(p) \subseteq M_i^+(Procs - \{p\})$. Thus, by Definition 2.4 we conclude this lemma holds. ■

A.2 Full proof for Lemma 2.7 (Round Progression Lemma)

Claim A.2. For any round $\bar{t}$ and processor $p \in Procs$, $R_{\overline{t+1}}\left(p\right) \cap M_{\bar{t}}^+\left(p\right) = \emptyset$.

Proof. For the purpose of contradiction, assume $R_{\overline{t+1}}\left(p\right) \cap M_{\bar{t}}^+\left(p\right) \neq \emptyset$. Thus, there is a step $j \in \left\{\bar{t}\left[0\right], \ldots, \bar{t}\left[C-1\right]\right\}$ such that $R_{\overline{t+1}}\left(p\right) \cap M_j^+\left(p\right) \neq \emptyset$. For such step j, let $S = R_{\overline{t+1}}\left(p\right) \cap M_j^+\left(p\right)$. Then,

$$\begin{aligned}
S &= R_{\overline{t+1}[0]}\left(p\right) \cap M_j^+\left(p\right) \\
&= R_{\overline{t+1}[0]}\left(p\right) \cap \left(R_j\left(p\right) \cap \left(R_{j+1} - R_{j+1}\left(p\right)\right)\right) \\
&= R_{\overline{t+1}[0]}\left(p\right) \cap R_j\left(p\right) \cap R_{j+1} \cap \overline{R_{j+1}\left(p\right)}
\end{aligned}$$

If j were $\bar{t}\left[C-1\right]$, then $S = \emptyset$, and so, as one can deduce, $j < \bar{t}\left[C-1\right]$. Now, consider a node μ such that $\mu \in R_{\overline{t+1}[0]}\left(p\right) \cap R_j\left(p\right) \cap R_{j+1} \cap \overline{R_{j+1}\left(p\right)}$, or, in other words, consider some $\mu \in S$. Since a node that is **ready** can only become **executed**, and a node in state **executed** does not change its state, it follows $\forall i \in \left\{j, \ldots, \overline{t+1}\left[0\right]\right\}, \mu \in R_i$. Moreover, as $\mu \in \overline{R_{j+1}\left(p\right)} \cap R_{s_l}\left(p\right)$ and $j+1 < \bar{t}\left[C-1\right]$, it follows that there is a step $k \in \left\{j+1, \ldots, \bar{t}\left[C-1\right]\right\}$ such that $\mu \in R_{k+1}\left(p\right) \cap \overline{R_k\left(p\right)} \cap R_k$. By Definition 2.2, it follows $\mu \in M_k^-\left(p\right)$, implying $\mu \in M_{\bar{t}}^-\left(p\right)$. However, since $\mu \in M_{\bar{t}}^-\left(p\right)$ and $\mu \in M_{\bar{t}}^+\left(p\right)$, it follows $M_{\bar{t}}^-\left(p\right) \cap M_{\bar{t}}^+\left(p\right) \neq \emptyset$, which, together with Lemma 2.5, contradicts Definition 2.3 — the definition of rounds. ■

Lemma A.3. *For any step i and $p \in Procs$, $R_{i+1}\left(p\right) \subseteq R_i\left(p\right) \cup E_i\left(p\right) \cup M_i^-\left(p\right)$.*

Proof of Lemma A.3. By Definition 2.2, it follows

$$\begin{aligned}
&R_i\left(p\right) \cup E_i\left(p\right) \cup M_i^-\left(p\right) \\
&= R_i\left(p\right) \cup \left(R_{i+1}\left(p\right) - R_i\right) \cup \left(R_{i+1}\left(p\right) \cap \left(R_i - R_i\left(p\right)\right)\right) \\
&= R_i\left(p\right) \cup \left(R_{i+1}\left(p\right) \cap \overline{R_i}\right) \cup \left(R_{i+1}\left(p\right) \cap R_i \cap \overline{R_i\left(p\right)}\right)
\end{aligned}$$

$$
\begin{aligned}
&= R_i\left(p\right) \cup \left[R_{i+1}\left(p\right) \cap \left(\overline{R_i} \cup \left(R_i \cap \overline{R_i\left(p\right)}\right)\right)\right] \\
&= \left[R_i\left(p\right) \cup R_{i+1}\left(p\right)\right] \cap \left[R_i\left(p\right) \cup \overline{R_i} \cup \left(R_i \cap \overline{R_i\left(p\right)}\right)\right] \\
&= \left[R_i\left(p\right) \cup R_{i+1}\left(p\right)\right] \cap \left[\left(R_i\left(p\right) \cup \overline{R_i}\right) \cup \left(R_i \cap \overline{R_i\left(p\right)}\right)\right] \\
&= \left[R_i\left(p\right) \cup R_{i+1}\left(p\right)\right] \cap \left[\left(R_i \cup \left(R_i\left(p\right) \cup \overline{R_i}\right)\right) \cap \left(\left(R_i\left(p\right) \cup \overline{R_i}\right) \cup \overline{R_i\left(p\right)}\right)\right] \\
&= R_i\left(p\right) \cup R_{i+1}\left(p\right)
\end{aligned}
$$

■

Lemma A.4. *For any steps s_0, s_1, with $s_1 > s_0$, and processor $p \in Procs$,*

$$\left[\bigcup_{i \in \{s_0,\ldots,s_1\}} R_i\left(p\right)\right] \subseteq R_{s_0}\left(p\right) \cup \left[\bigcup_{i \in \{s_0,\ldots,s_1-1\}} E_i\left(p\right) \cup M_i^-\left(p\right)\right].$$

Proof of Lemma A.4. Prove this lemma by induction.

Base case For the base case, let $s_1 = s_0 + 1$. Then,

$$\left[\bigcup_{i \in \{s_0,\ldots,s_1\}} R_i\left(p\right)\right] \subseteq R_{s_0}\left(p\right) \cup \left[\bigcup_{i \in \{s_0,\ldots,s_1-1\}} E_i\left(p\right) \cup M_i^-\left(p\right)\right]$$

iff $R_{s_0}\left(p\right) \cup R_{s_1}\left(p\right) \subseteq R_{s_0}\left(p\right) \cup E_{s_0}\left(p\right) \cup M_{s_0}^-\left(p\right)$. Taking into account Lemma A.3, we conclude the base case holds.

Induction step Assume that the result holds for some $s_l > s_0$, and show that it also holds for $s_l + 1$. The induction hypothesis is

$$\left[\bigcup_{i \in \{s_0,\ldots,s_l\}} R_i\left(p\right)\right] \subseteq R_{s_0}\left(p\right) \cup \left[\bigcup_{i \in \{s_0,\ldots,s_l-1\}} E_i\left(p\right) \cup M_i^-\left(p\right)\right]$$

and prove

$$
\begin{aligned}
\left[\bigcup_{i \in \{s_0,\ldots,s_l+1\}} R_i\left(p\right)\right] &\subseteq R_{s_0}\left(p\right) \cup \left[\bigcup_{i \in \{s_0,\ldots,(s_l+1)-1\}} E_i\left(p\right) \cup M_i^-\left(p\right)\right] \\
&= R_{s_0}\left(p\right) \cup \left[\bigcup_{i \in \{s_0,\ldots,s_l-1\}} E_i\left(p\right) \cup M_i^-\left(p\right)\right] \cup E_{s_l}\left(p\right) \cup M_{s_l}^-\left(p\right) \\
&\supseteq \left[\bigcup_{i \in \{s_0,\ldots,s_l\}} R_i\left(p\right)\right] \cup E_{s_l}\left(p\right) \cup M_{s_l}^-\left(p\right)
\end{aligned}
$$

Again, taking into account Lemma A.3, it is easy to deduce that the induction hypothesis holds, implying the lemma holds.

■

Lemma A.5. *For any round $\overline{t}$ and processor $p \in Procs$*

$$R_{\overline{t+1}}\left(p\right) \subseteq \left(E_{\overline{t}}\left(p\right) \cup R_{\overline{t}}\left(p\right) \cup M_{\overline{t}}^-\left(p\right)\right) - \left(C_{\overline{t}}\left(p\right) \cup M_{\overline{t}}^+\left(p\right)\right)$$

Proof of Lemma A.5. By Definitions 2.1, 2.2 and 2.4, it follows $R_{\overline{t+1}}\left(p\right) \cap C_{\overline{t}}\left(p\right) = \emptyset$. Since by Claim A.2, $R_{\overline{t+1}}\left(p\right) \cap M_{\overline{t}}^+\left(p\right) = \emptyset$, it follows $R_{\overline{t+1}}\left(p\right) \cap \left(C_{\overline{t}}\left(p\right) \cup M_{\overline{t}}^+\left(p\right)\right) = \emptyset$. Thus, it suffices to show that $R_{\overline{t+1}}\left(p\right) \subseteq E_{\overline{t}}\left(p\right) \cup R_{\overline{t}}\left(p\right) \cup M_{\overline{t}}^-\left(p\right)$. To conclude this proof, note that Lemma A.4, with $s_0 = \overline{t}\left[0\right]$ and $s_1 = \overline{t+1}\left[0\right]$, implies just that. ■

Claim A.6. For any round $\bar{t}$ and processor $p \in \mathit{Procs}$, $R_{\overline{t+1}}(p) \supseteq M^-_{\bar{t}}(p) - \left(C_{\bar{t}}(p) \cup M^+_{\bar{t}}(p)\right)$.

Proof. First, for an arbitrary step s_0 prove by induction that for any step s_1 such that $s_1 > s_0$,

$$R_{s_1}(p) \supseteq \left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} C_i(p) \cup M^+_i(p)\right]$$

Base case For the base case, let $s_1 = s_0 + 1$. Then,

$$R_{s_1}(p) \supseteq \left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} C_i(p) \cup M^+_i(p)\right]$$

iff $R_{s_1}(p) \supseteq M^-_{s_0}(p) - \left(C_{s_0}(p) \cup M^+_{s_0}(p)\right)$. To conclude the proof of the base case, note that

$$\begin{aligned}
& M^-_{s_0}(p) - \left(C_{s_0}(p) \cup M^+_{s_0}(p)\right) \\
& \subseteq M^-_{s_0}(p) \\
& = R_{s_1}(p) \cap \left(R_{s_0} - R_{s_0}(p)\right) \\
& \subseteq R_{s_1}(p)
\end{aligned}$$

Induction step Assume that the result holds for some $s_l > s_0$, and show that it also holds for $s_l + 1$. The induction hypothesis is

$$R_{s_l}(p) \supseteq \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i(p) \cup M^+_i(p)\right]$$

and we prove

$$\begin{aligned}
R_{s_{l+1}}(p) \supseteq & \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i(p) \cup M^+_i(p)\right] \\
= & \left(M^-_{s_l}(p) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\right]\right) - \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i(p) \cup M^+_i(p)\right] \\
= & \left(M^-_{s_l}(p) - \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i(p) \cup M^+_i(p)\right]\right) \\
& \cup \left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i(p) \cup M^+_i(p)\right]\right) \\
\subseteq & M^-_{s_l}(p) \cup \left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\right] - \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i(p) \cup M^+_i(p)\right]\right) \\
= & M^-_{s_l}(p) \cup \left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\right]\right. \\
& \left. - \left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i(p) \cup M^+_i(p)\right] \cup C_{s_l}(p) \cup M^+_{s_l}(p)\right)\right)
\end{aligned}$$

$$\begin{aligned}
&= M^-_{s_l}(p) \cup \Bigg(\Bigg(\Bigg[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^-_i(p)\Bigg] \\
&\qquad - \Bigg[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i(p) \cup M^+_i(p)\Bigg]\Bigg) - \left(C_{s_l}(p) \cup M^+_{s_l}(p)\right)\Bigg) \\
&\subseteq M^-_{s_l}(p) \cup \left(R_{s_l}(p) - \left(C_{s_l}(p) \cup M^+_{s_l}(p)\right)\right) \\
&= \left(R_{s_l+1}(p) \cap \left(R_{s_l} - R_{s_l}(p)\right)\right) \cup \left(R_{s_l}(p) - \left(C_{s_l}(p) \cup M^+_{s_l}(p)\right)\right) \\
&\subseteq R_{s_l+1}(p) \cup \left(R_{s_l}(p) - \left(C_{s_l}(p) \cup M^+_{s_l}(p)\right)\right) \\
&= R_{s_l+1}(p) \cup \left(R_{s_l}(p) \cap \overline{C_{s_l}(p)} \cap \overline{M^+_{s_l}(p)}\right) \\
&= R_{s_l+1}(p) \cup \left(R_{s_l}(p) \cap \left(\overline{R_{s_l}(p) - R_{s_l+1}}\right) \cap \left(\overline{R_{s_l}(p) \cap \left(R_{s_l+1} - R_{s_l+1}(p)\right)}\right)\right) \\
&= R_{s_l+1}(p) \cup \left(R_{s_l}(p) \cap \left(\overline{R_{s_l}(p)} \cup R_{s_l+1}\right) \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p)\right)\right) \\
&= R_{s_l+1}(p) \cup \left[\left(\left(R_{s_l}(p) \cap \overline{R_{s_l}(p)}\right) \cup \left(R_{s_l}(p) \cap R_{s_l+1}\right)\right) \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p)\right)\right] \\
&= R_{s_l+1}(p) \cup \left[R_{s_l}(p) \cap R_{s_l+1} \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p)\right)\right] \\
&= R_{s_l+1}(p) \cup \left[R_{s_l+1} \cap \left(\left(R_{s_l}(p) \cap \overline{R_{s_l}(p)}\right) \cup \left(R_{s_l}(p) \cap \overline{R_{s_l+1}}\right) \cup \left(R_{s_l}(p) \cap R_{s_l+1}(p)\right)\right)\right] \\
&\subseteq R_{s_l+1}(p) \cup \left[R_{s_l+1} \cap \left(\left(R_{s_l}(p) \cap \overline{R_{s_l+1}}\right) \cup R_{s_l+1}(p)\right)\right] \\
&\subseteq R_{s_l+1}(p) \cup \left[\left(R_{s_l+1} \cap R_{s_l}(p) \cap \overline{R_{s_l+1}}\right) \cup \left(R_{s_l+1} \cap R_{s_l+1}(p)\right)\right] \\
&\subseteq R_{s_l+1}(p) \cup \left(R_{s_l+1} \cap R_{s_l+1}(p)\right) \\
&= R_{s_l+1}(p)
\end{aligned}$$

To conclude this proof, let $s_0 = \bar{t}\,[0]$ and $s_1 = \overline{t+1}\,[0]$. ■

Claim A.7. For any steps s_0, s_1 such that $s_1 > s_0$:

$$R_{s_1}(p) \cup \Bigg[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} C_i(p)\Bigg] \supseteq \Bigg[\bigcap_{i\in\{s_0,\ldots,s_1-1\}} R_{s_0}(p) \cap \left(\overline{R_i(p)} \cup \overline{R_{i+1}} \cup R_{i+1}(p)\right)\Bigg]$$

Proof. Prove this claim for an arbitrary s_0 by induction on s_1.

Base case For the base case, consider $s_1 = s_0 + 1$. Then

$$R_{s_1}(p) \cup \Bigg[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} C_i(p)\Bigg] \supseteq \Bigg[\bigcap_{i\in\{s_0,\ldots,s_1-1\}} R_{s_0}(p) \cap \left(\overline{R_i(p)} \cup \overline{R_{i+1}} \cup R_{i+1}(p)\right)\Bigg]$$

iff

$$R_{s_1}(p) \cup C_{s_0}(p) \supseteq R_{s_0}(p) \cap \left(\overline{R_{s_0}(p)} \cup \overline{R_{s_0+1}} \cup R_{s_0+1}(p)\right)$$

To conclude the proof of the base case, note that

$$\begin{aligned}
&R_{s_0}(p) \cap \left(\overline{R_{s_0}(p)} \cup \overline{R_{s_0+1}} \cup R_{s_0+1}(p)\right) \\
&= \left(R_{s_0}(p) \cap \overline{R_{s_0}(p)}\right) \cup \left(R_{s_0}(p) \cap \overline{R_{s_1}}\right) \cup \left(R_{s_0}(p) \cap R_{s_1}(p)\right) \\
&\subseteq C_{s_0}(p) \cup R_{s_1}(p)
\end{aligned}$$

Induction step Assuming the claim holds for $s_l \geq s_0 + 1$, prove the claim also holds for $s_l + 1$.

Since by the induction hypothesis

$$R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \supseteq \left[\bigcap_{i \in \{s_0, \ldots, s_l - 1\}} R_{s_0}(p) \cap \left(\overline{R_i(p)} \cup \overline{R_{i+1}} \cup R_{i+1}(p) \right) \right]$$

it suffices to show that

$$\begin{aligned} R_{s_l+1}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} C_i(p) \right] \supseteq & \left(R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \right) \\ & \cap \left[R_{s_0}(p) \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p) \right) \right] \end{aligned}$$

To conclude,

$$\begin{aligned} & \left(R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \right) \cap \left[R_{s_0}(p) \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p) \right) \right] \\ & \subseteq \left(R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \right) \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p) \right) \\ & = \left(\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \cap \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p) \right) \right) \\ & \quad \cup \left(\left(R_{s_l}(p) \cap \overline{R_{s_l}(p)} \right) \cup \left(R_{s_l}(p) \cap \overline{R_{s_l+1}} \right) \cup \left(R_{s_l}(p) \cap R_{s_l+1}(p) \right) \right) \\ & \subseteq \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \cup C_{s_l}(p) \cup R_{s_l+1}(p) \\ & \subseteq R_{s_l+1}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \end{aligned}$$

■

Claim A.8. For any steps s_0, s_1 such that $s_1 > s_0$,

$$R_{s_1}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} C_i(p) \right] \supseteq \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} E_i(p) \right] - \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} M_i^+(p) \right]$$

Proof. Prove this claim for an arbitrary s_0 by induction on s_1.

Base case For the base case, consider $s_1 = s_0 + 1$. Then

$$R_{s_1}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} C_i(p) \right] \supseteq \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} E_i(p) \right] - \left[\bigcup_{i \in \{s_0, \ldots, s_1 - 1\}} M_i^+(p) \right]$$

iff

$$R_{s_1}(p) \cup C_{s_0}(p) \supseteq E_{s_0}(p) - M_{s_0}^+(p)$$

To conclude the proof of the base case, note that by Definition 2.2

$$\begin{aligned} & E_{s_0}(p) - M_{s_0}^+(p) \\ & = \left(R_{s_0+1}(p) - R_{s_0} \right) - \left(R_{s_0}(p) \cap \left(R_{s_0+1} - R_{s_0+1}(p) \right) \right) \\ & \subseteq R_{s_1}(p) \end{aligned}$$

Induction step Assuming the claim is true for $s_l \geq s_0 + 1$ show that it holds for $s_l + 1$. Thus, using the induction hypothesis

$$R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \supseteq \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} E_i(p) \right] - \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right]$$

show that

$$R_{s_l+1}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} C_i(p) \right] \supseteq \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} E_i(p) \right] - \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} M_i^+(p) \right]$$

It follows

$$\begin{aligned}
&\left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} E_i(p) \right] - \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} M_i^+(p) \right] \\
&= \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} E_i(p) \right] \cap \overline{\left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} M_i^+(p) \right]} \\
&= \left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} E_i(p) \right] \cap \left[\bigcap_{i \in \{s_0, \ldots, (s_l+1) - 1\}} \overline{M_i^+(p)} \right] \\
&= \overline{M_{s_l}^+(p)} \cap \left(\left[\bigcup_{i \in \{s_0, \ldots, (s_l+1) - 1\}} E_i(p) \right] \cap \left[\bigcap_{i \in \{s_0, \ldots, s_l - 1\}} \overline{M_i^+(p)} \right] \right) \\
&= \overline{M_{s_l}^+(p)} \cap \left(\left(E_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} E_i(p) \right] \right) \cap \left[\bigcap_{i \in \{s_0, \ldots, s_l - 1\}} \overline{M_i^+(p)} \right] \right) \\
&= \overline{M_{s_l}^+(p)} \cap \left(\left(E_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} E_i(p) \right] \right) \cap \overline{\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right]} \right) \\
&= \overline{M_{s_l}^+(p)} \cap \left[\left(E_{s_l}(p) \cap \overline{\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right]} \right) \right. \\
&\qquad \left. \cup \left(\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} E_i(p) \right] \cap \overline{\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right]} \right) \right] \\
&\subseteq \overline{M_{s_l}^+(p)} \cap \left[\left(E_{s_l}(p) \cap \overline{\left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right]} \right) \cup \left(R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \right) \right] \\
&\subseteq \left(\overline{R_{s_l}(p)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}(p) \right) \\
&\qquad \cap \left[\left(\left(R_{s_l+1}(p) - R_{s_l} \right) - \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} M_i^+(p) \right] \right) \cup \left(R_{s_l}(p) \cup \left[\bigcup_{i \in \{s_0, \ldots, s_l - 1\}} C_i(p) \right] \right) \right]
\end{aligned}$$

$$\begin{aligned}
&\subseteq \left(\overline{R_{s_l}\left(p\right)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}\left(p\right)\right) \cap \left(R_{s_l+1}\left(p\right) \cup R_{s_l}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right) \\
&= \left(R_{s_l}\left(p\right) \cup R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right) \cap \left(\overline{R_{s_l}\left(p\right)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}\left(p\right)\right) \\
&= \left[\left(R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right) \cap \left(\overline{R_{s_l}\left(p\right)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}\left(p\right)\right)\right] \\
&\qquad \cup \left[R_{s_l}\left(p\right) \cap \left(\overline{R_{s_l}\left(p\right)} \cup \overline{R_{s_l+1}} \cup R_{s_l+1}\left(p\right)\right)\right] \\
&\subseteq R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right] \\
&\qquad \cup \left[\left(R_{s_l}\left(p\right) \cap \overline{R_{s_l}\left(p\right)}\right) \cup \left(R_{s_l}\left(p\right) \cap \overline{R_{s_l+1}}\right) \cup \left(R_{s_l}\left(p\right) \cap R_{s_l+1}\left(p\right)\right)\right] \\
&\subseteq R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right] \cup \left(R_{s_l}\left(p\right) - R_{s_l+1}\right) \cup R_{s_l+1}\left(p\right) \\
&= R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right] \cup C_{s_l}\left(p\right) \\
&= R_{s_l+1}\left(p\right) \cup \left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i\left(p\right)\right]
\end{aligned}$$

■

Lemma A.9. *For any round $\bar{t}$ and processor $p \in Procs$*

$$R_{\overline{t+1}}\left(p\right) \supseteq \left(E_{\bar{t}}\left(p\right) \cup R_{\bar{t}}\left(p\right) \cup M^-_{\bar{t}}\left(p\right)\right) - \left(C_{\bar{t}}\left(p\right) \cup M^+_{\bar{t}}\left(p\right)\right)$$

Proof of Lemma A.9. To prove this direction of the lemma, it suffices to show:

1. $R_{\overline{t+1}}\left(p\right) \supseteq M^-_{\bar{t}}\left(p\right) - \left(C_{\bar{t}}\left(p\right) \cup M^+_{\bar{t}}\left(p\right)\right)$
2. $R_{\overline{t+1}}\left(p\right) \supseteq R_{\bar{t}}\left(p\right) - \left(C_{\bar{t}}\left(p\right) \cup M^+_{\bar{t}}\left(p\right)\right)$
3. $R_{\overline{t+1}}\left(p\right) \supseteq E_{\bar{t}}\left(p\right) - \left(C_{\bar{t}}\left(p\right) \cup M^+_{\bar{t}}\left(p\right)\right)$

Prove each of these propositions:

1. Claim A.6 implies Proposition 1 holds.

2. To prove Proposition 2:

$$\begin{aligned}
R_{\overline{t+1}}(p) &\supseteq R_{\overline{t}}(p) - \left(C_{\overline{t}}(p) \cup M_{\overline{t}}^{+}(p)\right)\\
&= \left(R_{\overline{t}}(p) - M_{\overline{t}}^{+}(p)\right) - C_{\overline{t}}(p)\\
&= \left(R_{\overline{t}[0]}(p) - \left[\bigcup_{i\in\left\{\overline{t}[0],\ldots,\overline{t}[C-1]\right\}} M_{i}^{+}(p)\right]\right) - C_{\overline{t}}(p)\\
&= \left(R_{\overline{t}[0]}(p) - \left[\bigcup_{i\in\left\{\overline{t}[0],\ldots,\overline{t}[C-1]\right\}} R_{i}(p) \cap \left(R_{i+1} - R_{i+1}(p)\right)\right]\right) - C_{\overline{t}}(p)\\
&= \left(R_{\overline{t}[0]}(p) \cap \left[\bigcap_{i\in\left\{\overline{t}[0],\ldots,\overline{t}[C-1]\right\}} \overline{R_{i}(p) \cap \left(R_{i+1} - R_{i+1}(p)\right)}\right]\right) - C_{\overline{t}}(p)\\
&= \left(R_{\overline{t}[0]}(p) \cap \left[\bigcap_{i\in\left\{\overline{t}[0],\ldots,\overline{t}[C-1]\right\}} \overline{R_{i}(p)} \cup \overline{R_{i+1}} \cup R_{i+1}(p)\right]\right) - C_{\overline{t}}(p)\\
&= \left[\bigcap_{i\in\left\{\overline{t}[0],\ldots,\overline{t}[C-1]\right\}} R_{\overline{t}[0]}(p) \cap \left(\overline{R_{i}(p)} \cup \overline{R_{i+1}} \cup R_{i+1}(p)\right)\right] - C_{\overline{t}}(p)
\end{aligned}$$

By Claim A.7, letting $s_0 = \overline{t}[0]$ and $s_1 = \overline{t+1}[0]$, it follows

$$R_{\overline{t+1}}(p) \supseteq \left(R_{\overline{t+1}}(p) \cup C_{\overline{t}}(p)\right) - C_{\overline{t}}(p) \supseteq \left(R_{\overline{t}}(p) - M_{\overline{t}}^{+}(p)\right) - C_{\overline{t}}(p).$$

3. By Claim A.8, letting $s_0 = \overline{t}[0]$ and $s_1 = \overline{t+1}[0]$, it follows

$$R_{\overline{t+1}}(p) \cup C_{\overline{t}}(p) \supseteq E_{\overline{t}}(p) - M_{\overline{t}}^{+}(p)$$

To conclude this proof note that

$$R_{\overline{t+1}}(p) \supseteq \left(R_{\overline{t+1}}(p) \cup C_{\overline{t}}(p)\right) - C_{\overline{t}}(p) \supseteq \left(E_{\overline{t}}(p) - M_{\overline{t}}^{+}(p)\right) - C_{\overline{t}}(p)$$

■

Proof of Lemma 2.7. Lemmas A.5 and A.9 imply this result. ■

A.3 Full proof for Lemma 2.10

Proof of Lemma 2.10. Since $R_{\overline{t}}$ is partitioned through the processors, by Requirement 2.6, $C_{\overline{t}}$ is also partitioned through the processors. Thus, $\sum_{p\in S}\left|R_{\overline{t}}(p) - C_{\overline{t}}(p)\right| = \left|R_{\overline{t}}(S) - C_{\overline{t}}(S)\right|$, and by the linearity of expectation $\sum_{p\in S} E\left[\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right|\right] = E\left[\left|R_{\overline{t+1}}(S) - C_{\overline{t+1}}(S)\right|\right]$. ■

A.4 Full proof for Lemma 2.13 (Connecting Lemma)

Claim A.10. For any round $\overline{t}$ and $p \in \mathit{Procs}$, $E_{\overline{t}}(p) \cap R_{\overline{t}}(p) = \emptyset$.

Proof. Given an arbitrary step s_0, we prove by induction on a step s_1 (where $s_1 > s_0$) that $\left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} E_i(p)\right] \cap R_{s_0}(p) = \emptyset$.

Base case Let $s_1 = s_0 + 1$. Then $\left[\bigcup_{i\in\{s_0,\ldots,s_1-1\}} E_i(p)\right] \cap R_{s_0}(p) = \emptyset$ iff $E_{s_0}(p) \cap R_{s_0}(p) = \emptyset$. By

Definition 2.2, it follows

$$\begin{aligned}
&E_{s_0}(p) \cap R_{s_0}(p) \\
&= (R_{s_0+1}(p) - R_{s_0}) \cap R_{s_0}(p) \\
&= R_{s_1}(p) \cap \overline{R_{s_0}} \cap R_{s_0}(p) \\
&\subseteq \overline{R_{s_0}} \cap R_{s_0} \\
&= \emptyset.
\end{aligned}$$

Induction step To prove the induction step, assume the lemma holds for a step $s_l > s_0$ and then prove that it also holds for s_{l+1}.

$$\begin{aligned}
&\left[\bigcup_{i \in \{s_0, \ldots, (s_l+1)-1\}} E_i(p)\right] \cap R_{s_0}(p) \\
&= \left(\left[\bigcup_{i \in \{s_0, \ldots, s_l-1\}} E_i(p)\right] \cap R_{s_0}(p)\right) \cup \left(E_{s_l}(p) \cap R_{s_0}(p)\right) \\
&= E_{s_l}(p) \cap R_{s_0}(p)
\end{aligned}$$

Since $s_l > s_0$, by Definition 2.1 it follows $E_{s_l} \cap R_{s_0} = \emptyset$, implying $E_{s_l}(p) \cap R_{s_0}(p) = \emptyset$.

To conclude the proof, let $s_0 = \bar{t}[0]$ and $s_1 = \overline{t+1}[0]$. ■

Claim A.11. For any round $\bar{t}$ and $p \in \mathit{Procs}$, $(R_{\bar{t}}(p) \cup E_{\bar{t}}(p)) \cap M^-_{\bar{t}}(p) = \emptyset$.

Proof. For the purpose of contradiction, let us assume $(R_{\bar{t}}(p) \cup E_{\bar{t}}(p)) \cap M^-_{\bar{t}}(p) \neq \emptyset$. Then, there must be a step $j \in \left\{\bar{t}[0], \ldots, \bar{t}[C-1]\right\}$ such that $(R_{\bar{t}}(p) \cup E_{\bar{t}}(p)) \cap M^-_j(p) \neq \emptyset$. Thus, at least one of the following propositions has to hold:

1. $R_{\bar{t}}(p) \cap M^-_j(p) \neq \emptyset$;

2. $E_{\bar{t}}(p) \cap M^-_j(p) \neq \emptyset$.

To conclude the proof of this claim, we prove that none of the propositions holds, contradicting our hypothesis that $(R_{\bar{t}}(p) \cup E_{\bar{t}}(p)) \cap M^-_{\bar{t}}(p) \neq \emptyset$.

Contradiction for Proposition 1 Let $S = R_{\bar{t}}(p) \cap M^-_j(p)$. Then,

$$\begin{aligned}
S &= R_{\bar{t}}(p) \cap M^-_j(p) \\
&= R_{\bar{t}[0]}(p) \cap (R_{j+1}(p) \cap (R_j - R_j(p))) \\
&= R_{\bar{t}[0]}(p) \cap R_{j+1}(p) \cap R_j \cap \overline{R_j(p)}
\end{aligned}$$

If j were $\bar{t}[0]$, then $S = \emptyset$, and so, $j > \bar{t}[0]$. Consider any node $\mu \in S$. Since a node that is **ready** can only become **executed**, and a node in state **executed** does not change its state, it follows $\forall i \in \left\{\bar{t}[0], \ldots, j+1\right\}, \mu \in R_i$. Noting that $\mu \in R_{\bar{t}[0]}(p) \cap \overline{R_j(p)} \cap R_j$, then there must be a step $k \in \left\{\bar{t}[0], \ldots, j-1\right\}$ such that $\mu \in R_k(p) \cap \overline{R_{k+1}(p)}$. Because $\forall i \in \left\{\bar{t}[0], \ldots, j+1\right\}, \mu \in R_i$, it follows $\mu \in R_k(p) \cap \overline{R_{k+1}(p)} \cap R_{k+1}$. By Definition 2.2, it follows $\mu \in M^+_k(p)$, implying $\mu \in M^+_{\bar{t}}(p)$. However, since $\mu \in M^-_{\bar{t}}(p)$ and $\mu \in M^+_{\bar{t}}(p)$, it follows $M^-_{\bar{t}}(p) \cap M^+_{\bar{t}}(p) \neq \emptyset$, which, together with Lemma 2.5, contradicts Definition 2.3 — the definition of a round.

Contradiction for Proposition 2 If $E_{\bar{t}}(p) \cap M_j^-(p) \neq \emptyset$, then there must be a step $m \in \left\{\bar{t}[0], \ldots, \bar{t}[C-1]\right\}$ such that $E_m(p) \cap M_j^-(p) \neq \emptyset$. Let $S = E_m(p) \cap M_j^-(p)$. It follows

$$\begin{aligned} S &= E_m(p) \cap M_j^-(p) \\ &= (R_{m+1}(p) - R_m) \cap (R_{j+1}(p) \cap (R_j - R_j(p))) \\ &= R_{m+1}(p) \cap \overline{R_m} \cap R_{j+1}(p) \cap R_j \cap \overline{R_j(p)} \end{aligned}$$

Consider any node $\mu \in S$.

If a node is not in state **ready** at step m, then it is either in state **not ready** or **executed**. Thus, at step m, μ is either in state **not ready** or **executed**. Because at step $m+1$ μ is in state **ready**, and since a node that is in state **executed** does not change its state, we deduce that μ is in state **not ready** at step m. Definition 2.1 then implies that until step m (including m), μ has been in state **not ready**.

By Definition 2.2, a node can only be migrated at some step i if it is ready at step i. Since μ is migrated at step j, then it must be ready at that step, implying $m < j$. Furthermore, if $m = j-1$, then $S = \emptyset$, and so, it follows $m \in \left\{\bar{t}[0], \ldots, j-2\right\}$.

Since a node that is **ready** can only become **executed**, and a node in state **executed** does not change its state, $\mu \in S$ implies $\forall i \in \{m+1, \ldots, j+1\}, \mu \in R_i$. Moreover, as $\mu \in R_{m+1}(p) \cap \overline{R_j(p)}$ and $m < j-1$, it follows that there is a step $k \in \{m+1, \ldots, j\}$ such that $\mu \in R_k(p) \cap \overline{R_{k+1}(p)}$. Because $\forall i \in \{m+1, \ldots, j+1\}, \mu \in R_i$, it follows $\mu \in R_k(p) \cap \overline{R_{k+1}(p)} \cap R_{k+1}$.

By Definition 2.2, it follows $\mu \in M_k^+(p)$, implying $\mu \in M_{\bar{t}}^+(p)$. However, since $\mu \in M_{\bar{t}}^-(p)$ and $\mu \in M_{\bar{t}}^+(p)$, it follows $M_{\bar{t}}^-(p) \cap M_{\bar{t}}^+(p) \neq \emptyset$, which, together with Lemma 2.5, contradicts Definition 2.3 — the definition of rounds.

■

Claim A.12. For any round $\bar{t}$ and $p \in \mathit{Procs}$, $C_{\bar{t}}(p) \cap M_{\bar{t}}^+(p) = \emptyset$.

Proof. Let $s_0 = \bar{t}[0]$. We prove by induction on a step $s_1 \in \left\{\bar{t}[0]+1, \ldots, \bar{t}[C-1]+1\right\}$ that $\left[\bigcup_{i \in \{s_0, \ldots, s_1-1\}} C_i(p)\right] \cap \left[\bigcup_{i \in \{s_0, \ldots, s_1-1\}} M_i^+(p)\right] = \emptyset$.

Base case Let $s_1 = s_0 + 1$. Then $\left[\bigcup_{i \in \{s_0, \ldots, s_1-1\}} C_i(p)\right] \cap \left[\bigcup_{i \in \{s_0, \ldots, s_1-1\}} M_i^+(p)\right] = \emptyset$ iff $C_{s_0}(p) \cap M_{s_0}^+(p) = \emptyset$. By Definition 2.2, it follows

$$\begin{aligned} & C_{s_0}(p) \cap M_{s_0}^+(p) \\ &= (R_{s_0}(p) - R_{s_1}) \cap (R_{s_0}(p) \cap (R_{s_1} - R_{s_1}(p))) \\ &= R_{s_0}(p) \cap \overline{R_{s_1}} \cap R_{s_0}(p) \cap R_{s_1} \cap \overline{R_{s_1}(p)} \\ &= \emptyset. \end{aligned}$$

Induction step To prove the induction step, assume the lemma holds for a step $s_l > s_0$ and then prove that it also holds for $s_l + 1$, where $(s_l + 1) \in \left\{\bar{t}[0]+1, \ldots, \bar{t}[C-1]+1\right\}$. The induction hypothesis is

$$\left[\bigcup_{i \in \{s_0, \ldots, s_l-1\}} C_i(p)\right] \cap \left[\bigcup_{i \in \{s_0, \ldots, s_l-1\}} M_i^+(p)\right] = \emptyset.$$

Thus,

$$\begin{aligned}
&\left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} C_i\left(p\right)\right]\cap\left[\bigcup_{i\in\{s_0,\ldots,(s_l+1)-1\}} M_i^+\left(p\right)\right]\\
&=\left(C_{s_l}\left(p\right)\cup\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)\cap\left(M_{s_l}^+\left(p\right)\cup\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\right)\\
&=\left(C_{s_l}\left(p\right)\cap\left(M_{s_l}^+\left(p\right)\cup\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\right)\right)\\
&\qquad\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\cap\left(M_{s_l}^+\left(p\right)\cup\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\right)\right)\\
&=\left(M_{s_l}^+\left(p\right)\cap C_{s_l}\left(p\right)\right)\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\cap C_{s_l}\left(p\right)\right)\\
&\qquad\cup\left(M_{s_l}^+\left(p\right)\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)\\
&\qquad\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\right)\\
&=\left(M_{s_l}^+\left(p\right)\cap C_{s_l}\left(p\right)\right)\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\cap C_{s_l}\left(p\right)\right)\\
&\qquad\cup\left(M_{s_l}^+\left(p\right)\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)\\
&=\left(\left(R_{s_l}\left(p\right)-R_{s_l+1}\right)\cap\left(R_{s_l}\left(p\right)\cap\left(R_{s_l+1}-R_{s_l+1}\left(p\right)\right)\right)\right)\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\cap C_{s_l}\left(p\right)\right)\\
&\qquad\cup\left(M_{s_l}^+\left(p\right)\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)\\
&=\left(R_{s_l}\left(p\right)\cap\overline{R_{s_l+1}}\cap R_{s_l}\left(p\right)\cap R_{s_l+1}\cap\overline{R_{s_l+1}\left(p\right)}\right)\cup\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\cap C_{s_l}\left(p\right)\right)\\
&\qquad\cup\left(M_{s_l}^+\left(p\right)\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)\\
&=\left(\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M_i^+\left(p\right)\right]\cap C_{s_l}\left(p\right)\right)\cup\left(M_{s_l}^+\left(p\right)\cap\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i\left(p\right)\right]\right)
\end{aligned}$$

By Definition 2.2, for any step i, $C_i(p)$ is composed by nodes that are attached to p at step i (implying they are **ready** at step i), but are no longer in state **ready** at step $i+1$. Thus, by Definition 2.1, since a node that is in state **ready** can only change its state to **executed**, and since a node that is **executed** can not become **not ready** nor **ready**, it follows $\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i(p)\right] \subseteq \mathit{Executed}_{s_l}$ (the set of nodes in state **executed** at step s_l). On the other hand, by Definition 2.2, a node can only be migrated from p at step s_l if it is ready at that step, implying $M^+_{s_l}(\mathit{Procs}) \subseteq R_{s_l}$. With this, because a node can only be in one state at each step, it follows $\mathit{Executed}_{s_l} \cap R_{s_l} = \emptyset$, implying $\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} C_i(p)\right] \cap M^+_{s_l}(p) = \emptyset$. As such, to conclude this proof it only remains to show that $\left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^+_i(p)\right] \cap C_{s_l}(p) = \emptyset$.

Let $S = \left[\bigcup_{i\in\{s_0,\ldots,s_l-1\}} M^+_i(p)\right] \cap C_{s_l}(p)$. For the purpose of contradiction, let us assume $S \neq \emptyset$. Thus, there is a step $j \in \{s_0,\ldots,s_l-1\}$ such that $M^+_j(p) \cap C_{s_l}(p) \neq \emptyset$. Let μ be some node such that $\mu \in M^+_j(p) \cap C_{s_l}(p)$. By Definition 2.2, it follows

$$\begin{aligned}
&M^+_j(p) \cap C_{s_l}(p)\\
&= \left(R_j(p) \cap \left(R_{j+1} - R_{j+1}(p)\right)\right) \cap \left(R_{s_l}(p) - R_{s_l+1}\right)\\
&= R_j(p) \cap R_{j+1} \cap \overline{R_{j+1}(p)} \cap R_{s_l}(p) \cap \overline{R_{s_l+1}}
\end{aligned}$$

which implies $\mu \in R_j(p) \cap R_{j+1} \cap \overline{R_{j+1}(p)} \cap R_{s_l}(p) \cap \overline{R_{s_l+1}}$. If j were s_l-1, it would follow $R_j(p) \cap R_{j+1} \cap \overline{R_{j+1}(p)} \cap R_{s_l}(p) \cap \overline{R_{s_l+1}} = \emptyset$, and so $j < s_l - 1$. Since a node that is **ready** can only become **executed**, and a node in state **executed** does not change its state, then $\forall i \in \{j,\ldots,s_l-1\}, \mu \in R_i$. Moreover, as $\mu \in \overline{R_{j+1}(p)} \cap R_{s_l}(p)$ and $s_l > j+1$, it follows that there is a step $k \in \{j+1,\ldots,s_l-1\}$ such that $\mu \in R_{k+1}(p) \cap \overline{R_k(p)} \cap R_k$. By Definition 2.2, it follows $\mu \in M^-_k(p)$, implying $\mu \in M^-_{\bar{t}}(p)$. However, since $\mu \in M^-_{\bar{t}}(p)$ and $\mu \in M^+_{\bar{t}}(p)$, it follows $M^-_{\bar{t}}(p) \cap M^+_{\bar{t}}(p) \neq \emptyset$, which, together with Lemma 2.5, contradicts Definition 2.3 (specifically, that no node is migrated more than once during the same round).

■

Claim A.13. For any round $\bar{t}$ and $p \in \mathit{Procs}$, $R_{\bar{t}}(p) \cup E_{\bar{t}}(p) \cup M^-_{\bar{t}}(p) \supseteq C_{\bar{t}}(p) \cup M^+_{\bar{t}}(p)$.

Proof. By Requirement 2.6, it follows $R_{\bar{t}}(p) \cup E_{\bar{t}}(p) \cup M^-_{\bar{t}}(p) \supseteq C_{\bar{t}}(p)$. Thus, it suffices to show that $R_{\bar{t}}(p) \cup E_{\bar{t}}(p) \cup M^-_{\bar{t}}(p) \supseteq M^+_{\bar{t}}(p)$. By Definition 2.2, for any step i, $M^+_i(p) = R_i(p) \cap (R_{i+1} - R_{i+1}(p))$, implying $R_i(p) \supseteq M^+_i(p)$. To conclude this proof, let $s_0 = \bar{t}[0]$ and $s_1 = \bar{t}[C-1]$ in Lemma A.4. ■

Proof of Lemma 2.13. First, note that $|E_{\bar{t}}(p)| < \left|C_{\overline{t+1}}(p)\right| + \left|M^+_{\bar{t}}(p)\right|$ iff

$$\begin{aligned}
|E_{\bar{t}}(p)| + |R_{\bar{t}}(p)| + \left|M^-_{\bar{t}}(p)\right| - |C_{\bar{t}}(p)| - \left|M^+_{\bar{t}}(p)\right| &< \left|C_{\overline{t+1}}(p)\right| + \left|M^+_{\bar{t}}(p)\right| + |R_{\bar{t}}(p)| + \left|M^-_{\bar{t}}(p)\right|\\
&\quad - |C_{\bar{t}}(p)| - \left|M^+_{\bar{t}}(p)\right|\\
&= \left|C_{\overline{t+1}}(p)\right| + |R_{\bar{t}}(p)| + \left|M^-_{\bar{t}}(p)\right| - |C_{\bar{t}}(p)|
\end{aligned}$$

Noting that:

1. Claim A.10 implies
$$|E_{\bar{t}}(p)| + |R_{\bar{t}}(p)| = |E_{\bar{t}}(p) \cup R_{\bar{t}}(p)|;$$
2. Claim A.11 implies
$$|R_{\bar{t}}(p) \cup E_{\bar{t}}(p)| + \left|M^-_{\bar{t}}(p)\right| = \left|R_{\bar{t}}(p) \cup E_{\bar{t}}(p) \cup M^-_{\bar{t}}(p)\right|;$$

3. Claim A.12 implies
$$\left|C_{\overline{t}}(p)\right| + \left|M^{+}_{\overline{t}}(p)\right| = \left|C_{\overline{t}}(p) \cup M^{+}_{\overline{t}}(p)\right|;$$

4. Claim A.13 implies
$$\left|E_{\overline{t}}(p) \cup R_{\overline{t}}(p) \cup M^{-}_{\overline{t}}(p)\right| - \left|C_{\overline{t}}(p) \cup M^{+}_{\overline{t}}(p)\right| = \left|\left(E_{\overline{t}}(p) \cup R_{\overline{t}}(p) \cup M^{-}_{\overline{t}}(p)\right) - \left(C_{\overline{t}}(p) \cup M^{+}_{\overline{t}}(p)\right)\right|; \text{and}$$

5. Lemma 2.7 implies
$$\left|R_{\overline{t+1}}(p)\right| = \left|\left(E_{\overline{t}}(p) \cup R_{\overline{t}}(p) \cup M^{-}_{\overline{t}}(p)\right) - \left(C_{\overline{t}}(p) \cup M^{+}_{\overline{t}}(p)\right)\right|,$$

it follows $\left|R_{\overline{t+1}}(p)\right| = \left|E_{\overline{t}}(p)\right| + \left|R_{\overline{t}}(p)\right| + \left|M^{-}_{\overline{t}}(p)\right| - \left|C_{\overline{t}}(p)\right| - \left|M^{+}_{\overline{t}}(p)\right|$. To conclude this proof, note that Requirement 2.6 implies $\left|R_{\overline{t}}(p)\right| - \left|C_{\overline{t}}(p)\right| = \left|R_{\overline{t}}(p) - C_{\overline{t}}(p)\right|$ and $\left|R_{\overline{t+1}}(p)\right| - \left|C_{\overline{t+1}}(p)\right| = \left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right|$, and so, it follows
$$\left|R_{\overline{t+1}}(p)\right| < \left|C_{\overline{t+1}}(p)\right| + \left|R_{\overline{t}}(p)\right| + \left|M^{-}_{\overline{t}}(p)\right| - \left|C_{\overline{t}}(p)\right|$$
$$\text{iff}$$
$$\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right| < \left|R_{\overline{t}}(p) - C_{\overline{t}}(p)\right| + \left|M^{-}_{\overline{t}}(p)\right|.$$

■

B 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 [8]. 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 [8] has been proven to satisfy the relaxed semantics on any good set of invocations. Note that any set of invocations made during the execution of a computation scheduled by either WS or 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.

C Full proofs for the results obtained in Section 3.2

C.1 Full proof for Lemma 3.5

Lemma C.1. *For WS, at any round $\bar{t}$, $M^{+}_{\bar{t}}(p) = Stolen^{+}_{\bar{t}}(p)$ and $M^{-}_{\bar{t}}(p) = Stolen^{-}_{\bar{t}}(p)$.*

Proof of Lemma C.1. Both results follow from Definition 3.4 and Algorithm 1. ■

Lemma C.2. *Suppose there are B bins and $B.\alpha$ balls, and that each ball is tossed independently and uniformly at random into the bins. For a 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, $E[Y_i] = P\{Y_i = 1\} \geq 1 - e^{-\alpha}$.

Proof of Lemma C.2. The probability that no ball lands in b_i is $P\{Y_i = 0\} = \left(1 - \frac{1}{B}\right)^{B\alpha} \leq e^{-\alpha}$. To conclude, $E[Y_i] = P\{Y_i = 1\} \geq 1 - e^{-\alpha}$. ■

Lemma C.3. *For any round $\bar{t}$ and $p \in U_{\bar{t}}$ during a computation's execution using WS, we have $1 - e^{-\alpha_{\bar{t}}} \leq \mathrm{E}[|Stolen^{+}_{\bar{t}}(p)|] \leq \alpha_{\bar{t}}$.*

Proof of Lemma C.3. 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 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 C.2 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 $(P\alpha_{\bar{t}})/P = \alpha_{\bar{t}}$. Since p may only invoke the *popBottom* method of its deque during the second phase and the all the steal attempts take place during the first phase, then, taking into account the deque semantics (see Section B): 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, $\mathrm{E}[|Stolen^{+}_{\bar{t}}(p)|] \geq 1 - e^{-\alpha_{\bar{t}}}$ and $\mathrm{E}[|Stolen^{+}_{\bar{t}}(p)|] \leq \alpha_{\bar{t}}$. ■

Proof of Lemma 3.5. Lemmas C.1 and C.3 imply this result. ■

C.2 Full proof for Lemma 3.6

Proof of Lemma 3.6. By the definition of Algorithm 1 it can be proved by induction on the progression of a computation's execution that if a processor has at least one attached node at the beginning of round $\bar{t}$, then the processor executes a node during $\bar{t}$. From that, and by observing the algorithm, it follows that if p has at least one node attached, then it does not make any steal attempt during $\bar{t}$, implying $Stolen^{-}_{\bar{t}}(p) = \emptyset$. Lemma C.1 then implies $M^{-}_{\bar{t}}(p) = \emptyset$. On the other hand, since p always executes one of its attached nodes if there is any, it follows that if $p \in U_{\bar{t}}$ then p's deque is not empty.

If p only has a single attached node, then $p \in S_{\bar{t}}$. Because it has one attached node, it follows $Stolen^{-}_{\bar{t}}(p) = \emptyset$. Again, Lemma C.1 then implies $M^{-}_{\bar{t}}(p) = \emptyset$. In addition, since the out-degree of any node is at most two (by our conventions regarding computations' structure), then at the end of the round p has at most two attached nodes.

Finally we show that if $R_{\bar{t}}(p) = \emptyset$, then at the end of the round p has at most one attached node. If p has no attached node, then its *assigned* variable does not contain a valid node, implying that p executes a call to the $WorkMigration$ procedure. Since each call only entails one invocation to the *popTop* method, then, taking into account the method's semantics (see Section B) it follows that p may only get at most one node from its steal attempt. Since after performing such attempt, p takes no further action during the scheduling iteration other than simply waiting for it to end, we conclude the lemma holds. ■

C.3 Full proof for Theorem 3.7

Proof of Theorem 3.7. Due to our conventions related with the computations' structure, it follows that during a round a processor can enable two nodes. For some round $\bar{t}$, let p be a non-self-stable processor (*i.e.* $p \in U_{\bar{t}}$) such that $|E_{\bar{t}}(p)| = 2$. Lemmas 2.7 and 3.6 imply $R_{\overline{t+1}}(p) = (E_{\bar{t}}(p) \cup R_{\bar{t}}(p)) - (C_{\bar{t}}(p) \cup M^{+}_{\bar{t}}(p))$. As already noted in the proof of Lemma 3.6, for the WS algorithm, since $p \in U_{\bar{t}}$, it follows that $|C_{\bar{t}}(p)| = 1$. Since we have 1. $E_{\bar{t}}(p) \cup R_{\bar{t}}(p) \supseteq C_{\bar{t}}(p) \cup M^{+}_{\bar{t}}(p)$; 2. $E_{\bar{t}}(p) \cap R_{\bar{t}}(p) = \emptyset$; and 3. $C_{\bar{t}}(p) \cap M^{+}_{\bar{t}}(p) = \emptyset$ [3], then $\left|R_{\overline{t+1}}(p)\right| = |E_{\bar{t}}(p)| + |R_{\bar{t}}(p)| - |C_{\bar{t}}(p)| - |M^{+}_{\bar{t}}(p)|$. By Lemma 3.5 and since p enabled two nodes, it follows $\mathrm{E}[\left|R_{\overline{t+1}}(p)\right|] \geq 2 + |R_{\bar{t}}(p)| - 1 - \alpha_{\bar{t}} = |R_{\bar{t}}(p)| + 1 - \alpha_{\bar{t}}$. Since p enabled two nodes, it executes a node during the next round, implying $\left|C_{\overline{t+1}}(p)\right| = 1$. It then follows, $\mathrm{E}[\left|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)\right|] \geq |R_{\bar{t}}(p)| + 1 - \alpha_{\bar{t}} - 1 = |R_{\bar{t}}(p)| - \alpha_{\bar{t}}$. Even though the definition of *algorithm short-term stability* only considers ratios of idle processors in $]0;1[$, note that for WS, if all processors are idle, then the computation's execution must have already finished, and so it only makes sense to analyze rounds during which the execution is still ongoing. It then follows that $\alpha_{\bar{t}} < 1$, implying $\mathrm{E}[|R_{\overline{t+1}}(p) - C_{\overline{t+1}}(p)|] \geq |R_{\bar{t}}(p)| - \alpha_{\bar{t}} > |R_{\bar{t}}(p) - C_{\bar{t}}(p)|$. Thus, if during round $\bar{t}$, p were the only non-self-stable processor (*i.e.* $U_{\bar{t}} = \{p\}$), then $\mathrm{E}[|R_{\overline{t+1}}(U_{\bar{t}}) - C_{\overline{t+1}}(U_{\bar{t}})|] > |R_{\bar{t}}(U_{\bar{t}}) - C_{\bar{t}}(U_{\bar{t}})|$. ■

As one might note, this implies that WS can not even guarantee *short-term stability* for the set $\{p\}$ (recall Definition 2.9), regardless of the ratio of idle processors.

D Full proofs for the results obtained in Section 4

D.1 Proof for Claim 4.2

Proof of Claim 4.2. As one can observe from Algorithm 3, like WS, WSS can also be naturally defined using scheduling loops (lines 2 to 24): 1. at most one node is executed per scheduling iteration; 2. if a node is migrated to a processor during an iteration, then it is not migrated again; 3. the length of every iteration is bounded by a constant; and 4. the full sequence of instructions executed by any processor can be partitioned into a sequence of scheduling iterations. As for WS, and for the same reasons, we do not formally show that the loop starting at line 2 and ending at line 25 of Algorithm 3 satisfies the requirements of a scheduling loop. Nevertheless, it is easy to deduce, by observing the definition of the scheduler (given in Algorithm 3), that this claim holds. ■

D.2 Full proof for Theorem 4.3

Lemma D.1. *For WSS, at any round $\bar{t}$ and for any processor p, $M^{+}_{\bar{t}}(p) = Stolen^{+}_{\bar{t}}(p) \cup Spread^{+}_{\bar{t}}(p)$ and $M^{-}_{\bar{t}}(p) = Stolen^{-}_{\bar{t}}(p) \cup Spread^{-}_{\bar{t}}(p)$.*

Proof of Lemma D.1. Both results follow from Definition 3.4 and Algorithm 3. ■

Lemma D.2. *For WSS, at any round $\bar{t}$, $Stolen^{+}_{\bar{t}}(p) \cap Spread^{+}_{\bar{t}}(p) = \emptyset$.*

Proof of Lemma D.2. As proved in Claim A.10, $R_{\bar{t}}(p) \cap E_{\bar{t}}(p) = \emptyset$. To conclude the proof of this lemma, note that, from Definitions 3.4 and 4.1, and by the definition of Algorithm 3 we have $Stolen^{+}_{\bar{t}}(p) \subseteq R_{\bar{t}}(p)$ and $Spread^{+}_{\bar{t}}(p) \subseteq E_{\bar{t}}(p)$ and so the lemma holds. ■

[3] For a formal proof of these claims see Claims A.13, A.10 and A.12 for parts 1, 2 and 3, respectively, with $M^{-}_{\bar{t}}(p) = \emptyset$.

Lemma D.3. *Consider some $p \in Procs$ and some round $\overline{t}$ during the execution of a computation by WSS. Then: 1. if $p \in U_{\overline{t}}$ then p's deque is non-empty and $M^{-}_{\overline{t}}(p) = \emptyset$; and 2. if $p \in S_{\overline{t}}$ then $\left|R_{\overline{t+1}}(p)\right| \leq 2$.*

Proof of Lemma D.3. This proof follows the same general arguments as the proof of Lemma 3.6.

As for WS, taking into account the definition of the WSS scheduler (depicted in Algorithm 3) it can be proved by induction on the progression of the computation's execution that if a processor has at least one attached node at the beginning of round $\overline{t}$, then the processor executes a node during $\overline{t}$. From that, and by observing the algorithm, it follows that if p has at least one node attached, then: 1. p does not make any steal attempt during $\overline{t}$, implying $Stolen^{-}_{\overline{t}}(p) = \emptyset$; and 2. p's *state* flag is set to WORKING at least until the beginning of the third stage of round $\overline{t}$, implying that no processor donates work to p, and so $Spread^{-}_{\overline{t}}(p) = \emptyset$. Thus, taking into account Lemma D.1, if p has at least one attached node then $M^{-}_{\overline{t}}(p) = \emptyset$.

To conclude the proof of the first statement of this lemma, note that because p always executes one of its attached nodes as long as there is any, it follows that if $p \in U_{\overline{t}}$ then p has at least two nodes attached and so p's deque can not be empty.

Again, since p executes one of its attached nodes as long as there is any, if p only has a single attached node then $p \in S_{\overline{t}}$ and $M^{-}_{\overline{t}}(p) = \emptyset$. By the nodes' out-degree assumption, it then follows that at the end of round $\overline{t}$, p can have at most two attached nodes.

Finally we show that if $R_{\overline{t}}(p) = \emptyset$, then at the end of the round p has at most two attached nodes. If p has no attached node, then its *assigned* variable does not contain a valid node, implying that p executes a call to the *LoadBalance* procedure (line 22). Since each call only entails one invocation to the *popTop* method, then, taking into account the method's semantics (see Section B) it follows that p may only get at most one node from its steal attempt. On the other hand, as one can deduce from the definition of the *LoadBalance* procedure, p can only accept at most one node donation during the procedure's invocation[4]. Thus, at the end of the call p can only have at most two attached nodes. To conclude the proof of the second part of the lemma, note that, after the call to the *LoadBalance* procedure returns, p takes no further action during the iteration. ■

Recall that C denotes the length of each round, and that by definition $C \geq 1$. Then, from Lemma D.3, it follows that $\forall p \in S_{\overline{t}}, \left|R_{\overline{t+1}}(p)\right| \leq 2 \leq C+1$. Furthermore, taking into account Lemma 2.10 and Corollary 2.14, it follows that to prove this theorem it suffices to show that for any round $\overline{t}$ such that $\alpha_{\overline{t}} \in [0,7375;1[$, we have $\forall p \in U_{\overline{t}}, \quad \left|E_{\overline{t}}(p)\right| < \mathrm{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\overline{t}}(p)\right|]$.

Lemma D.4. *For any round $\overline{t}$ and $p \in U_{\overline{t}}$ during a computation's execution using WSS, $1 - e^{-\alpha_{\overline{t}}} \leq \mathrm{E}[|Stolen^{+}_{\overline{t}}(p)|] \leq \alpha_{\overline{t}}$.*

Proof of Lemma D.4. The proof of this result is identical to the proof of Lemma C.3, following from Lemma D.3 and the definition of WSS (see Algorithm 3). ■

Due to our conventions related with computations' structure, $|E_{\overline{t}}(p)|$ is either 0, 1, or 2:

- If $|E_{\overline{t}}(p)| = 0$ then, by Lemma D.4 it follows $|E_{\overline{t}}(p)| < \mathrm{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\overline{t}}(p)\right|]$.
- If $|E_{\overline{t}}(p)| = 1$ then, by the definition of WSS it follows $\left|C_{\overline{t+1}}(p)\right| = 1$. Taking into account Lemma D.4, we deduce $|E_{\overline{t}}(p)| < \mathrm{E}[\left|C_{\overline{t+1}}(p)\right| + \left|M^{+}_{\overline{t}}(p)\right|]$.
- By the definition of WSS it follows that if $|E_{\overline{t}}(p)| = 2$ then $\left|C_{\overline{t+1}}(p)\right| = 1$. Thus, to prove this case it suffices to show $1 < \mathrm{E}[\left|M^{+}_{\overline{t}}(p)\right|]$. 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 3.

[4] In fact, p only accepts a donation if its steal attempt failed, and so p can only have at most one attached node.

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 willing to accept a node are the ones who are idle and whose steal attempt (that took place during the first phase) failed. Furthermore, each such processor only accepts at most a single node.

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

Lemma D.5. *Let* $spreads\left(p,\alpha,d\right)$ *be a function corresponding to the expected number of nodes that* p *spreads during any round, where the ratio of idle processors of the round is* α *and the number of processors enabling two nodes is* d*. Then,* $spreads\left(p,\alpha,d\right) \geq spreads\left(p,\alpha,P\left(1-\alpha\right)\right)$*.*

Proof of Lemma D.5. 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 make a spread attempt. However, if p targets a processor whose *state* flag is set to IDLE, then its attempt has a chance to succeed. We now consider the two possible situations:

$d = P\left(1-\alpha\right)$ — In this case, $spreads\left(p,\alpha,d\right) = spreads\left(p,\alpha,P\left(1-\alpha\right)\right)$.

$d \neq P\left(1-\alpha\right)$ — By definition there are $P\left(1-\alpha\right)$ busy processors, implying that $d \leq P\left(1-\alpha\right)$. Thus, for this case we conclude $d < P\left(1-\alpha\right)$. Now, suppose p targets some processor q whose *state* flag is set to IDLE. Thanks to the synchronous environment we have artificially created, and assuming that any call to $UniformlyRandomNumber$ takes the same number of steps, then every processor executes the CAS instruction — whose success dictates the success of the spread attempt — at the same step (line 30 of Algorithm 3). Finally, as a consequence of our assumptions regarding the CAS instruction (see the first paragraph of Section 4) and since processors target donees uniformly at random, the greater the number of spread attempts that target q the smaller are the chances for p's spread attempt to succeed, concluding the proof of this lemma.

■

Lemma D.6. *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* $B\left(1-\alpha\right) = B_G$*. Now, suppose there are* B_R *cubes and* B_G *balls. First, each cube is tossed, independently and uniformly at random, into the bins. After tossing all the* B_R *cubes, count the number of cubes that landed in green bins, and, for each such cube, a red bin is painted green. After finishing all the paintings, each of the* B_G *balls is tossed, independently and uniformly at random, into the bins.*
Let Y *denote the number of bins that are still red, with at least one ball. Then,*

$$E\left[Y\right] \geq B\alpha^2\left(1-e^{-\left(1-\alpha\right)}\right).$$

Proof of Lemma D.6. 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

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

Similarly to Lemma C.2, for a bin b_i let Y_i be an indicator variable, defined as

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

Thus, the probability that none of the B_G balls lands in b_i is

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

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

$$P\left\{Y_i = 0 | B_{R\mapsto G} = m\right\} = P\left\{Y_i = 0\right\}, \quad \text{and,} \quad P\left\{Y_i = 1 | B_{R\mapsto G} = m\right\} = P\left\{Y_i = 1\right\}.$$

It follows

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

Consider

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

corresponding to the number of bins that are still red with at least one ball. By the linearity of expectation, it follows

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

To conclude this proof, by the law of total expectation it follows

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

■

We now obtain lower bounds on the total number of spreads (or donations) made to processors during the second phase of some scheduling iteration, assuming that all busy processors make a spread attempt.

Lemma D.7. *Consider any round $\bar{t}$ during a computation's execution, 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* $\mathrm{E}[|\mathit{Spread}^{+}_{\bar{t}}(B_{\bar{t}})|] \geq P\alpha^2_{\bar{t}}\left(1 - e^{-(1-\alpha_{\bar{t}})}\right)$.

Proof of Lemma D.7. We prove this result by making an analogy with Lemma D.6: 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 cube toss corresponds to a steal attempt; 4. 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 5. each ball toss corresponds to a spread attempt. Note that we can make this analogy because 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 of scheduling iterations. Thus, $\mathrm{E}[|\mathit{Spread}^{+}_{\bar{t}}(B_{\bar{t}})|] \geq P\alpha^2_{\bar{t}}\left(1 - e^{-(1-\alpha_{\bar{t}})}\right)$. ■

Lemma D.8. $\mathrm{E}[|\mathit{Spread}^{+}_{\bar{t}}(p)|] \geq \frac{\alpha^2_{\bar{t}}}{1-\alpha_{\bar{t}}}\left(1 - e^{-(1-\alpha_{\bar{t}})}\right)$.

Proof of Lemma D.8. By Lemma D.5 it follows that $\mathrm{E}[|\mathit{Spread}^{+}_{\bar{t}}(p_{\bar{t}})|]$ is the smallest if all busy processors enabled two nodes, and thus made a spread attempt. By Lemma D.7, the expected number of nodes spread during a round such that all busy processors make a spread attempt is at least $P\alpha^2_{\bar{t}}\left(1 - e^{-(1-\alpha_{\bar{t}})}\right)$. 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

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

■

Claim D.9. Let

$$v(\alpha) = \frac{-2 + e^{\alpha-1}\left(2 + \alpha\left(4 - 5\alpha + \alpha^3\right)\right)}{(\alpha-1)^3}.$$

Then, $\forall \alpha \in [0.7375; 1[\qquad v(\alpha) \geq 0$.

Proof. Let

$$f(\alpha) = -2 + e^{\alpha-1}\left(2 + \alpha\left(4 - 5\alpha + \alpha^3\right)\right)$$

and

$$g(\alpha) = (\alpha-1)^3.$$

Thus,

$$v(\alpha) = \frac{f(\alpha)}{g(\alpha)}.$$

Since

$$\frac{\mathrm{d}f(\alpha)}{\mathrm{d}\alpha} = e^{-1+\alpha}(1-\alpha)^2\left(6 + 6\alpha + \alpha^2\right)$$

it follows that $\forall \alpha \in [0.7375; 1[$, $f(\alpha)$ is non-decreasing.

Consequently, $\forall \alpha \in [0.7375; 1[$

$$\begin{aligned} f(\alpha) &\leq f(1) \\ &= -2 + e^{1-1}\left(2 + 1\left(4 - 5.1 + 1^3\right)\right) \\ &= 0 \end{aligned}$$

Since, $\forall \alpha \in [0.7375; 1[$

$$g(\alpha) = (-1+\alpha)^3 < 0$$

we have that, $\forall \alpha \in [0.7375; 1[$

$$v(\alpha) = \frac{f(\alpha)}{g(\alpha)} \geq 0,$$

concluding the proof of the claim. ■

Lemma D.10. $\forall \alpha \in [0, 7375; 1[, \quad 1 < 1 - e^{-\alpha} + \frac{\alpha^2}{1-\alpha}\left(1 - e^{-(1-\alpha)}\right).$

Proof of Lemma D.10.

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

iff

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

Let

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

Then

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

Let $t(\alpha)$ be defined as the last two terms of $\frac{\mathrm{d}s(\alpha)}{\mathrm{d}\alpha}$:

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

To prove that $t(\alpha)$ is non-decreasing $\forall \alpha \in [0.7375; 1[$, we compute its derivative.

$$\frac{\mathrm{d}t(\alpha)}{\mathrm{d}\alpha} = \frac{-2 + e^{\alpha-1}\left(2 + \alpha\left(4 - 5\alpha + \alpha^3\right)\right)}{(-1+\alpha)^3}$$

By Claim D.9, $\forall \alpha \in [0.7375; 1[$ we have $\frac{\mathrm{d}t(\alpha)}{\mathrm{d}\alpha} \geq 0$, meaning that $t(\alpha)$ is non-decreasing for that interval.

It follows that $\forall \alpha \in [0.7375; 1[$ we have

$$\begin{aligned} t(\alpha) &= \frac{1 + e^{-1+\alpha}\alpha\left(-2+\alpha^2\right)}{(-1+\alpha)^2} \\ &\geq \frac{1 + e^{-1+0.7375}0.7375\left(-2+0.7375^2\right)}{(-1+0.7375)^2} \\ &> 2.5 \end{aligned}$$

Consequently,

$$\begin{aligned}\frac{\mathrm{d}s\left(\alpha\right)}{\mathrm{d}\alpha} &= -1+e^{-\alpha}+\frac{1+e^{-1+\alpha}\alpha\left(-2+\alpha^{2}\right)}{\left(-1+\alpha\right)^{2}}\\ &= -1+e^{-\alpha}+t\left(\alpha\right)\\ &> e^{-\alpha}+2.5\\ &> 0\end{aligned}$$

Thus, $\forall\alpha\in[0.7375;1[$, $s\left(\alpha\right)$ is strictly increasing. To conclude this proof, it only remains to note that for that same interval we have

$$\begin{aligned}s\left(\alpha\right) &\geq s\left(0.7375\right)\\ &= -e^{-0.7375}+\frac{0.7375^{2}}{1-0.7375}\left(1-e^{-(1-0.7375)}\right)\\ &> 0.00006\\ &> 0\end{aligned}$$

■

By Lemmas D.2 and D.1, it follows $\left|M_{\bar{t}}^{+}\left(p\right)\right|=\left|Stolen_{\bar{t}}^{+}\left(p\right)\right|+\left|Spread_{\bar{t}}^{+}\left(p\right)\right|$, and by Lemmas D.4 and D.8, it follows $\mathrm{E}[\left|M_{\bar{t}}^{+}\left(p\right)\right|]\geq 1-e^{-\alpha_{\bar{t}}}+\frac{\alpha_{\bar{t}}^{2}}{1-\alpha_{\bar{t}}}\left(1-e^{-(1-\alpha_{\bar{t}})}\right)$. Thus, taking into account Lemma D.10, having $\alpha=\alpha_{\bar{t}}$, we conclude the proof of Theorem 4.3. ■