Fully Read/Write Fence-Free Work-Stealing with Multiplicity

Armando Castañeda
Institute of Mathematics, National Autonomous University of Mexico, Mexico City, Mexico

Miguel Piña
Faculty of Sciences, National Autonomous University of Mexico, Mexico City, Mexico

Abstract

It is known that any algorithm for *work-stealing* in the standard asynchronous shared memory model must use expensive Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions. There have been proposed algorithms for relaxations in the standard model and algorithms in restricted models that avoid the impossibility result, but only in some operations.

This paper considers work-stealing with *multiplicity*, a relaxation in which every task is taken by *at least* one operation, with the requirement that any process can extract a task *at most once*. Two versions of the relaxation are considered and two *fully* Read/Write algorithms are presented in the standard asynchronous shared memory model, both devoid of Read-After-Write synchronization patterns in all its operations, the second algorithm additionally being *fully fence-free*, namely, no specific ordering among the algorithm's instructions is required, beyond what is implied by data dependence. To our knowledge, these are the first algorithms for work-stealing possessing all these properties. Our algorithms are also wait-free solutions of relaxed versions of single-enqueue multi-dequeuer queues. The algorithms are obtained by reducing work-stealing with multiplicity and weak multiplicity to MaxRegister and RangeMaxRegister, a relaxation of MaxRegister which might be of independent interest. An experimental evaluation shows that our fully fence-free algorithm exhibits better performance than Cilk THE, Chase-Lev and Idempotent Work-Stealing algorithms.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Correctness condition, Linearizability, Nonblocking, Relaxed data type, Set-linearizability, Wait-freedom, Work-stealing

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.16

Related Version *Full Version*: `https://arxiv.org/abs/2008.04424` [8]

Supplementary Material *Software (Source Code)*: `https://bit.ly/2WHomWg`
archived at `swh:1:dir:a6173150e85d59f84d9840aff7f2bc92d194f858`

Funding *Armando Castañeda*: Supported by UNAM-PAPIIT project IN108720.
Miguel Piña: Recipient of a PhD fellowship from CONACyT.

1 Introduction

Context. *Work-stealing* is a popular technique to implement dynamic *load balancing* for efficient task parallelization of irregular workloads. It has been used in several contexts, from programming languages and parallel-programming frameworks to SAT solver and state space search exploration in model checking (e.g. [5, 7, 11, 14, 15, 24, 28]). In work-stealing, each process *owns* a set of tasks that have to be executed. The *owner* of the set can put tasks in it and can take tasks from it to execute them. When a process runs out of tasks (i.e. the set is empty), instead of being idle, it becomes a *thief* to steal tasks from a *victim*. Thus, a work-stealing algorithm provides three high-level operations: Put and Take, which can be invoked only by the owner, and Steal, which can be invoked by any thief.

A main target when designing work-stealing algorithms is to make Put and Take as simple and efficient as possible. Unfortunately, it has been shown that any work-stealing algorithm in the standard asynchronous shared memory model must use Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions (e.g. Compare&Swap or Test&Set) [4]. Read-After-Write synchronization patterns are based on the *flag principle* [20] (i.e. writing on a shared variable and then reading another variable), and hence when implementing an algorithm using such a synchronization pattern in real multicore architectures, a memory *fence* (also called *barrier*) is required so that the read and write instructions are not reordered by the compiler or the processor. It is well-known that fences that avoid reads and writes to be reordered are highly costly, while atomic Read-Modify-Write instructions, with high coordination power (which can be formally measured through the *consensus number* formalism [19]), are in principle slower than the simple Read/Write instructions.[1] Indeed, the known work-stealing algorithms in the literature are based on the flag principle in their Take/Steal operations [12, 15, 16, 17]. Thus, a way to circumvent the result in [4] is to consider work-stealing with relaxed semantics, or to make extra assumptions on the model. As far as we know, [25] and [26] are the only works that have followed these directions.

Observing that in some contexts it is ensured that no task is repeated (e.g. by checking first whether a task is completed) or the nature of the problem solved tolerates repeatable work (e.g. parallel SAT solvers), Michael, Vechev and Saraswat propose *idempotent* work-stealing [25], a relaxation allowing a task to be taken *at least once*, instead of *exactly once*. Three idempotent work-stealing algorithms are presented in [25], that insert/extract tasks in different orders. The relaxation allows each of the algorithms to circumvent the impossibility result in [4], only in its Put and Take operations as they use only Read/Write instructions and are devoid of Read-After-Write synchronization patterns; however, Steal uses Compare&Swap. Moreover, Put requires that some Write instructions are not reordered and Steal requires that some Read instructions are not reordered, and thus fences are required when the algorithms are implemented; fences between Read (resp. Write) instructions, however, are usually not too costly in practice. As for progress guarantees, Put and Take are *wait-free* while Steal is only *nonblocking*.

Morrison and Afek consider the restricted TSO model [29] and present two work-stealing algorithms in [26] whose Put operation is wait-free and uses only Read/Write instructions, and Take and Steal are either nonblocking and use Compare&Swap, or blocking and use a *lock*. The algorithms are clever adaptations of the well-known Cilk THE and Chase-Lev work-stealing algorithms [12, 15] to the TSO model. Generally speaking, in this model Write (resp. Read) instructions cannot be reordered, hence fences among Write (resp. Read) instructions are not needed; additionally, each process has a local buffer where its Write instructions are stored until they are eventually propagated to the main memory (in FIFO order). Reordering some Write (resp. Read) instructions of the algorithms in [26] compromises correctness, however TSO prevents this to happen. To avoid Read-After-Write patterns, it is considered in [26] that buffers are of bounded size.

Contributions. In this paper we are interested in the following theoretical question: whether there are meaningful relaxations of work-stealing that allow us to design fully Read/Write, fully wait-free and fully *fence-free* algorithms in the standard asynchronous shared memory model; fence-free means that algorithm's correctness does not require any specific instruction

[1] In practice, contention might be the dominant factor, namely, an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions.

ordering, beyond what is implied by data dependence. In other words, we are interested in knowing if simple synchronization mechanisms are suffice to solve non-trivial relaxations of work-stealing. This question has a practical motivation too, as [25] and [26] have shown that performance of work-stealing implementations can be increased by reducing the usage of Read-After-Write patterns and Read-Modify-Write instructions.

We consider work-stealing with *multiplicity* [10], a relaxation in which every task is taken by *at least* one operation, and, differently from idempotent work-stealing, it requires that if a task is taken by several Take/Steal operations, then they must be *pairwise concurrent*; therefore, no more than the number of processes in the system can take the same task. In our relaxation, tasks are inserted/extracted in FIFO order. We present a fully Read/Write algorithm for work-stealing with multiplicity, with its Put operation being fence-free and its Take and Steal operations being devoid of Read-After-Write synchronization patterns. As for progress, all operations are wait-free with Put having constant *step complexity* and Take and Steal having logarithmic step complexity. The algorithm stands for its simplicity, based on a single instance of a MaxRegister object [3, 23], hence showing that work-stealing with multiplicity reduces to MaxRegister.

We also consider a variation of work-stealing with multiplicity in which Take/Steal operations extracting the same task *may not be concurrent*, however, each process extracts a task *at most once*; this variant, which inserts/extracts tasks in FIFO order too, is called work-stealing with *weak multiplicity*. For this relaxation, we present an algorithm, inspired in our first solution, which uses only Read/Write instructions, is *fully fence-free* and all its operations are wait-free; furthermore each operation has constant step complexity. To our knowledge, this is the first algorithm for work-stealing having all these properties. The algorithm is obtained by reducing work-stealing with weak multiplicity to RangeMaxRegister, a relaxation of MaxRegister that we propose here, which might be of independent interest.

We show that each of our algorithms can be easily modified so that every task is extracted by a bounded number of operations, more specifically, by at most one Steal operation. In the modified algorithms, Put and Take remain the same, and a single Swap instruction is added to Steal. Also, the algorithms can be modified so that multiplicity is provided on demand, namely, a task can be taken by multiple operations only if indicated. These variants can be used in contexts where repeatable work is not allowed or needs to be restricted.

We stress that our algorithms are also wait-free solutions of relaxed versions of single-enqueue multi-dequeuer queues, namely, with multiplicity and weak multiplicity. To the best of our knowledge, together with idempotent FIFO, these are the only single-enqueue multi-dequeuer queue relaxations that have been studied. Formal specifications and correctness proofs are provided, using the *linearizability* [21] and *set-linearizable* correctness formalisms [9, 27]. Intuitively, set-linearizability is a generalization of linearizability in which several concurrent operations are allowed to be linearized at the same linearization point.

We complement our results by studying the question if there are implementations of our algorithms with good performance in practical settings. We conducted an experimental evaluation comparing our algorithms to Cilk THE, Chase-Lev and the idempotent work-stealing algorithms. In the experiments, our second algorithm exhibits a better performance than the previously mentioned algorithms, while its bounded version, with a Swap-based Steal operation, shows a lower performance but keeps competitive.

Work-stealing with multiplicity and idempotent work-stealing are closely related, but they are not the same. We observe that the idempotent work-stealing algorithms in [25] allow a task to be extracted by an unbounded number of Steal operations. This observation implies that the algorithms in [25] do not solve work-stealing with multiplicity. Therefore, the relaxations and algorithms proposed here provide stronger guarantees than idempotent work-stealing algorithms, without the need of heavy synchronization mechanisms.

Related Work. To the best of our knowledge, [25] and [26] are the only works that have been able to avoid costly synchronization mechanisms in work-stealing algorithms. The three idempotent algorithms in [25] insert/extract tasks in FIFO and LIFO orders, and as in a double-ended queue (the owner working on one side and the thieves on the other). As already mentioned, the algorithms in [26] are adaptations of the well-known Cilk THE and Chase-Lev work-stealing algorithms to the TSO model. Cilk THE and Chase-Lev (and other standard work-stealing algorithms) insert/extract tasks using double-ended queue, with the aim that the owner uses heavy synchronization mechanisms to coordinate with the thieves only when the queue is almost empty.

The experimental evaluation shows that the idempotent algorithms outperform Cilk THE and Chase-Lev. The work-stealing algorithms in [26] are fence-free adaptations of Cilk THE and Chase-Lev [12, 15] to the TSO model. The model itself guarantees that fences among Write (resp. Read) instructions are not needed, and the authors of [26] assume that write-buffers in TSO are bounded so that a fence-free coordination mechanism can be added to the Take and Steal operations of Cilk THE and Chase-Lev. The experimental evaluation in [26] shows that their algorithms outperform Cilk THE and Chase-Lev and sometimes achieve performance comparable to the idempotent work-stealing algorithms.

The notion of multiplicity was recently introduced in [10] for queues and stacks. In a queue/stack with multiplicity, an item can be dequeued/popped by several operations but only if they are concurrent. Read/Write set-linearizable algorithms for queues and stacks with multiplicity and without Read-After-Write synchronization patterns are provided in [10], and it is noted that these algorithms are also solutions for work-stealing with multiplicity; the step complexity of Enqueue/Push, which implements Put, is $\Theta(n)$ while the step complexity of Dequeue/Pop, which implements Take and Steal, is unbounded, where n denotes the number of processes. Our algorithms follow different principles than those in [10]. Our weak multiplicity relaxation of work-stealing follows the spirit of the relaxations in [10] in the sense that it requires that any algorithm for the relaxation provides "exact" responses in sequential executions, namely, the relaxation applies only if there is no contention.

As far we know, only two single-enqueue multi-dequeuer queue algorithms have been proposed [13, 22]. The algorithm in [13] is wait-free with constant step complexity and uses Read-Modify-Write instructions with consensus number 2, specifically, Swap and Fetch&Increment; however, the algorithm uses arrays of infinite length. It is explained in [13] that this assumption can be removed at the cost of increasing the step complexity to $O(n)$, where n denotes the number of processes. The algorithm in [22] is wait-free with $O(\log n)$ step complexity and uses LL/SC, a Read-Modify-Write instruction whose consensus number is ∞, and using a bounded amount of memory. Our results show that there are single-enqueue multi-dequeuer queue relaxations that can be solved using very light synchronization mechanisms.

2 Preliminaries

Model of Computation. We consider a standard concurrent shared memory system with $n \geq 2$ *asynchronous* processes, $p_0, \ldots, p_{n-1}$, which may *crash* at any time during an execution. Processes communicate with each other by invoking *atomic* instructions of *base* objects: either simple Read/Write instructions, or more powerful Read-Modify-Write instructions, such as Swap or Compare&Swap. The *index* of process p_i is i.

An *algorithm* for a high-level concurrent object T (e.g. a queue or a stack) is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which instructions of base objects p_i execute in order to return a response, when it invokes a

(high-level) operation of T; each of these instructions is a *step*. An *execution* of $\mathcal{A}$ is a (possibly infinite) sequence of steps, plus invocations and responses of operations of the concurrent object T; for any invocation of a process p_i, the steps of p_i between that invocation and its corresponding response (if there is one), denoted are steps that are specified by the local state machine A_i. An operation call in an execution is *complete* if both its invocation and response appear in the execution, otherwise it is *pending*.

A process is *correct* in an infinite execution if it takes infinitely many steps. An algorithm, or an operation of it, is *nonblocking* if whenever processes take steps, at least one of the invocations terminates, namely, in every infinite execution, infinitely many operations terminate [21]. An algorithm, or an operation of it, is *wait-free* if every process completes each operation in a finite number of its steps, namely, every correct process completes infinitely many operations [19]. *Bounded wait-freedom* [18] additionally requires that there is a bound on the number of steps needed to terminate. The *step complexity* of an operation is the maximum number steps a process needs to execute in order to return.

In a Read-After-Write synchronization pattern, a process first writes in a shared variable and then reads another shared variable, maybe executing other instructions in between. An algorithm, or one of its operations, is *fence-free* if it does not require any specific ordering among its steps, beyond what is implied by data dependence (e.g. the value written by a Write instruction depends on the value read by previous a Read instruction). Note that a fence-free algorithm does not use Read-After-Write synchronization patterns. In our algorithms, we use the notation $\{O_1.inst_1, \ldots, O_x.inst_x\}$ to denote that the instructions $O_1.inst_1, \ldots, O_x.inst_x$ can be executed in any order. Observe that fence instructions (also called memory barriers) are not required to correctly implement a fence-free algorithm in a concrete programming language or multicore architecture since any reordering of non-data-dependent instructions does not affect the correctness of the algorithm. For sake of simplicity in the analysis, first we will present our algorithms using base objects of infinite length and arrays of infinite length. Later we explain how we deal with this unrealistic assumptions.

Correctness Conditions. *Linearizability* [21] is the standard notion used to identify a correct implementation. Intuitively, an execution is linearizable if its (high-level) operations can be ordered sequentially, without reordering non-overlapping operations, so that their responses satisfy the specification of the implemented object. A *sequential specification* of a concurrent object T is a state machine specified through a transition function δ. Given a state q and an invocation $inv(\mathsf{op})$, $\delta(q, inv(\mathsf{op}))$ returns the tuple $(q', res(\mathsf{op}))$ (or a set of tuples if the machine is *non-deterministic*) indicating that the machine moves to state q' and the response to op is $res(\mathsf{op})$. The sequences of invocation-response tuples, $\langle inv(\mathsf{op}) : res(\mathsf{op})\rangle$, produced by the state machine are its *sequential executions*. Given an execution E, we write $\mathsf{op} <_E \mathsf{op}'$ if and only if $res(\mathsf{op})$ precedes $inv(\mathsf{op}')$ in E. Two operations are *concurrent*, denoted $\mathsf{op} \,||_E\, \mathsf{op}'$, if they are incomparable by $<_E$. The execution is *sequential* if $<_E$ is a total order.

▶ **Definition 1** (Linearizability). *Let $\mathcal{A}$ be an algorithm for a concurrent object T. A finite execution E of $\mathcal{A}$ is* linearizable *if there is a sequential execution S of T such that (1) S contains every completed operation of E and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in E, and (2) for every two completed operations op and op' in E, if $\mathsf{op} <_E \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is* linearizable *if each of its finite executions is linearizable.*

Intuitively, while linearizability requires a total order on the operations, set-linearizability [9, 27] allows several operations to be linearized at the same linearization point. A *set-sequential specification* of a concurrent object differs from a sequential execution in that δ

receives as input the current state q of the machine and a set $Inv = \{inv(\mathsf{op}_1), \ldots, inv(\mathsf{op}_t)\}$ of operation invocations that happen concurrently. Thus $\delta(q, Inv)$ returns (q', Res) where q' is the next state and $Res = \{res(\mathsf{op}_1), \ldots, res(\mathsf{op}_t)\}$ are the responses to the invocations in Inv (if the machine is non-deterministic, Res is a set of sets of responses). The sets Inv and Res are called *concurrency classes*. The sequences of invocation-response concurrency classes, $\langle INV : RES \rangle$, produced by the state machine are its *set-sequential executions*. Observe that in a sequential specification all concurrency classes have a single element.

Let $\mathcal{A}$ be an algorithm for a concurrent object T. A finite execution E of $\mathcal{A}$ is *set-linearizable* if there is a set-sequential execution S of T such that (1) S contains every completed operation of E and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in E, and (2) for every two completed operations op and op' in E, if $\mathsf{op} <_E \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is *set-linearizable* if each of its finite executions is set-linearizable.

3 Work-Stealing with Multiplicity

Work-stealing with *multiplicity* is a relaxation of the usual work-stealing in which every task is extracted *at least once*, and if it is extracted by several operations, they must be *concurrent*. In the formal set-sequential specification below (and in its variant in the next section), tasks are inserted/extracted in FIFO order but it can be easily adapted to encompass other orders (e.g. LIFO). Figure 1 depicts an example of a set-sequential execution of the work-stealing with multiplicity, where concurrent Take/Steal operations can extract the same task.

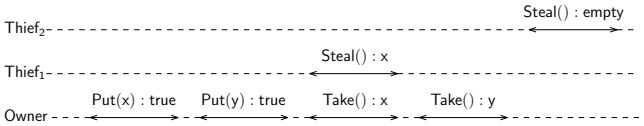


Figure 1 A set-seq. exec. of work-stealing with multiplicity.

▶ **Definition 2** ((FIFO) Work-Stealing with Multiplicity)**.** *The universe of tasks that the owner can put is* $\mathbf{N} = \{1, 2, \ldots\}$*, and the set of states* Q *is the infinite set of finite strings* $\mathbf{N}^*$*. The initial state is the empty string, denoted* ϵ*. In state* q*, the first element in* q *represents the* head *and the last one the* tail*.* $\forall q \in Q$*,* $0 \leq t \leq n-1$*,* $x \in \mathbf{N}$*, the transitions are:*

1. $\delta(q, \mathsf{Put}(x)) = (q \cdot x, \langle \mathsf{Put}(x) : \mathsf{true} \rangle)$.
2. $\delta(x \cdot q, \{\mathsf{Take}(), \mathsf{Steal}_1(), .., \mathsf{Steal}_t()\}) = (q, \{\langle \mathsf{Take}() : x \rangle, \langle \mathsf{Steal}_1() : x \rangle, .., \langle \mathsf{Steal}_t() : x \rangle\})$.
3. $\delta(x \cdot q, \{\mathsf{Steal}_1(), .., \mathsf{Steal}_t()\}) = (q, \{\langle \mathsf{Steal}_1() : x \rangle, .., \langle \mathsf{Steal}_t() : x \rangle\})$.
4. $\delta(\epsilon, \mathsf{Take}()) = (\epsilon, \langle \mathsf{Take}() : \mathsf{empty} \rangle)$.
5. $\delta(\epsilon, \mathsf{Steal}()) = (\epsilon, \langle \mathsf{Steal}() : \mathsf{empty} \rangle)$.

Let $\mathcal{A}$ be a linearizable algorithm for work-stealing with multiplicity. Note that items 2 and 3 in Definition 2 and the definition of set-linearizability directly imply that in every execution of $\mathcal{A}$, the number of Take/Steal operations that take the same task is at most the number of processes in the system, as the operations must be pairwise concurrent to be set-linearized together. Furthermore, every *sequential* execution of $\mathcal{A}$ looks like an "exact" solution for work-stealing, as every operation is linearized alone, by definition of set-linearizability; formally, every sequential execution of $\mathcal{A}$ is sequential execution of (FIFO) work-stealing. We call this property *sequentially-exact*. Thus, in the absence of contention, $\mathcal{A}$ provides an exact solution for work-stealing.

▶ Remark 3. Any set-lin. algorithm for work-stealing with multiplicity is sequentially-exact.

Work-Stealing with Multiplicity from MaxRegister. Here we show that work-stealing with multiplicity can be reduced to a single instance of a MaxRegister object (defined below). We will argue that our algorithm and the Read/Write wait-free MaxRegister algorithm in [3], provide a work-stealing a solution with multiplicity with logarithmic step complexity and without Read-After-Write synchronization patterns in Take and Steal.

```
Shared Variables:
    Head: atomic MaxRegister object initialized to 1
    Tasks[1, 2, ...]: array of atomic Read/Write objects
                with the first two objects initialized to ⊥
Persistent Local Variables of the Owner:
    tail ← 0

Operation Put(x):
(01) tail ← tail + 1
(02) {Tasks[tail].Write(x), Task[tail + 2].Write(⊥)}
(03) return true
end Put

Operation Take():
(04) head ← Head.MaxRead()
(05) if head ≤ tail then
(06)    {x ← Tasks[head].Read(), Head.MaxWrite(head + 1)}
(07)    return x
(08) return empty
end Take

Operation Steal():
(09) head ← Head.MaxRead()
(10) x ← Tasks[head].Read()
(11) if x ≠ ⊥ then
(12)    Head.MaxWrite(head + 1)
(13)    return x
(14) return empty
end Steal
```

Figure 2 WS-MULT: a MaxRegister-based set-lin. algorithm for work-stealing with multiplicity.

The algorithm presented in this section does not seem to have practical implications, however it will lead us to our efficient fully fence-free Read/Write work-stealing algorithm with constant step complexity in all its operations. Figure 2 presents WS-MULT, a set-linearizable algorithm for work-stealing with multiplicity. The algorithm is based on a single wait-free linearizable MaxRegister object, which provides two operations: MaxRead that returns the maximum value written so far in the object, and MaxWrite that writes a new value only if it is greater than the largest value that has been written so far. In WS-MULT, the tail of the queue is stored in the local persistent variable *tail* of the owner, while the head is stored in the shared MaxRegister *Head*. Recall that the notation in Line 2 denotes that the instructions can be executed in any order. The semantics of MaxRegister guarantees that *Head* contains the current value of the head at all times, as a "slow" process cannot "move back" the head by writing a smaller value in *Head*. Thus, the MaxRegister *Head* acts as a sort of barrier in the algorithm. The net effect of this is that the only way that two Take/Steal operations return the same task is because they are concurrent, reading the same value from *Head*.

Note that if only the first object in *Tasks* is initialized to $\perp$ (and hence Put is modified accordingly), it is possible that a thief reads a value from *Tasks* that has not been written by the owner: in an execution with a single Put(x) operation, the steps in Line 2 could be executed $Tasks[1]$.Write(x) first and then $Task[2]$.Write($\perp$) with a sequence of two Steal operations completing in between, hence the second operations reading $Tasks[2]$ which has not been written yet by the owner, which is a problem if $Tasks[2]$ contains a value distinct from non-$\perp$ value.

We also observe that the algorithm remains correct if Take is the same as Steal; Take in 2 saves a Read step when the queue is empty.

▶ **Theorem 4.** *Algorithm* WS-MULT *is a set-linearizable wait-free fence-free algorithm for work-stealing with multiplicity using atomic* Read/Write *objects and a single atomic* MaxRegister *object. All operations have constant step complexity and* Put *is fully* Read/Write.

When we replace $Head$ with the wait-free linearizable Read/Write MaxRegister algorithm in [3], whose step complexity is $O(\log m)$, where $m \geq 1$ is the maximum value that can be stored in the object, the step complexity of WS-MULT is bounded wait-free with logarithmic step complexity too. In the resulting algorithm at most m tasks can be inserted. Since the algorithm in [3] does not use Read-After-Write synchronization patterns (as explained in the proof of Theorem 5), the resulting algorithm does not use those patterns either.

▶ **Theorem 5.** *If $Head$ is an instance of the wait-free linearizable* Read/Write MaxRegister *algorithm in [3],* WS-MULT *is linearizable and fully* Read/Write *with* Take *and* Steal *having step complexity $O(\log m)$, where m denotes the maximum number or tasks that can be inserted in an execution.* Take *and* Steal *do not use* Read-After-Write *synchronization patterns.*

4 Work-Stealing with Weak Multiplicity

In the context of work-stealing, a logarithmic step complexity of the Take operation may be prohibitive in practice. Ideally, we would like to have constant step complexity, in all operations if possible, and using simple synchronization mechanisms. In this section, we propose work-stealing with *weak* multiplicity, which admits fully Read/Write fence-free implementations with constant step complexity in all its operations. Intuitively, weak multiplicity requires that every task is extracted at least once, but now every process extracts a task *at most once*, hence Take/Steal operations returning the same task *might not* be concurrent. Therefore, the relaxation retains the property that the number of operations that can extract the same task is at most the number of processes in the system.

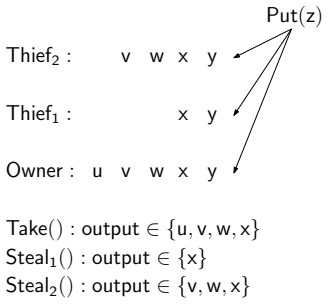


Figure 3 A schematic view of weak multiplicity.

Figure 3 depicts a schematic view of a state of work-stealing with weak multiplicity. Intuitively, at any state, each process has its own *virtual* queue of tasks. When the owner inserts a new task, it concurrently places the task in all virtual queues. Therefore, in any state, for any pair of process' virtual queues, one of them is suffix of the other. A Take/Steal operation can return any task from its virtual queue that is not "beyond" the first task of the *shortest* virtual queue in the state. In the example, a Steal operation of thief p_1, denoted Steal$_1$(), can return only x, as p_1 has the shortest virtual queue in the state, while a Take operation of the owner can return any task in $\{u, v, w, x\}$, which contains any task from the beginning of its virtual queue up to x. In our algorithms, the virtual queues are implemented using a single queue.

The next *sequential* specification formally defines work-stealing with weak multiplicity. Without loss of generality, the specification assumes that p_0 is the owner, and each invocation/response of thief p_i is subscripted with its index $i \in \{1, \ldots, n-1\}$. Figure 4 shows an example of a sequential execution of work-stealing with weak multiplicity; note that Take/Steal operations are allowed to get the same item, although they are not concurrent.

▶ **Definition 6** ((FIFO) Work-Stealing with Weak Multiplicity)**.** *The universe of tasks that the owner can put is* $\mathbf{N} = \{1, 2, \ldots\}$*, and the set of states* Q *is the infinite set of* n*-vectors of finite strings* $\mathbf{N}^* \times \ldots \times \mathbf{N}^*$*, with the property that for any two pairs of strings in a vector, one of them is a* suffix of the other*. The initial state is the vector with empty strings,* $(\epsilon, \ldots, \epsilon)$*.* $\forall (t_0, \ldots, t_{n-1}) \in Q$*, the transitions are the following:*

1. $\delta((t_0, \ldots, t_{n-1}), \mathsf{Put}(x)) = ((t_0 \cdot x, \ldots, t_{n-1} \cdot x), \langle \mathsf{Put}(x) : \mathsf{true} \rangle)$.
2. *If* $t_0 = x_1 \cdots x_j \cdot q \neq \epsilon$ *with* $j \geq 1$ *and* $x_j \cdot q$ *being the shortest string in the state (possibly with* $x_j \cdot q = \epsilon$*), then* $\delta((t_0, \ldots, t_{n-1}), \mathsf{Take}()) = \{((\widehat{t_0}, t_1, \ldots, t_{n-1}), \langle \mathsf{Take}() : x_k \rangle)\}$*, where* $k \in \{1, \ldots, j\}$ *and* $\widehat{t_0} = x_{k+1} \cdots x_j \cdot q$*.*
3. *If* $t_i = x_1 \cdots x_j \cdot q \neq \epsilon$ *with* $j \geq 1$*,* $i \in \{1, \ldots, n-1\}$ *and* $x_j \cdot q$ *being the shortest string in the state (possibly with* $x_j \cdot q = \epsilon$*), then* $\delta((t_0, \ldots, t_{n-1}), \mathsf{Steal}_i()) = \{((t_0, \ldots, t_{i-1}, \widehat{t_i}, t_{i+1}, \ldots, t_{n-1}), \langle \mathsf{Steal}_i() : x_k \rangle)\}$*, where* $k \in \{1, \ldots, j\}$ *and* $\widehat{t_i} = x_{k+1} \cdots x_j \cdot q$*.*
4. *If* $t_0 = \epsilon$*, then* $\delta((\epsilon, t_1, \ldots, t_{n-1}), \mathsf{Take}()) = ((\epsilon, t_1, \ldots, t_{n-1}), \langle \mathsf{Take}() : \epsilon \rangle)$.
5. *If* $t_i = \epsilon$ *with* $i \in \{1, \ldots, n-1\}$*, then* $\delta((t_0, \ldots, t_{i-1}, \epsilon, t_{i+1}, \ldots, t_{n-1}), \mathsf{Steal}_i()) = ((t_0, \ldots, t_{i-1}, \epsilon, t_{i+1}, \ldots, t_{n-1}), \langle \mathsf{Steal}_i() : \epsilon \rangle)$.

Observe that the second and third items in Definition 6 correspond to non-deterministic transitions in which a Take/Steal operation can extract any task of $\{x_1, \ldots, x_j\}$; the value returned can be ϵ when the shortest string in the state ($x_j \cdot q'$ in the definition) is ϵ. Furthermore, the definition guarantees that every task is extracted at least once because every Take/Steal operation can only return a task that is not "beyond" the first task in the shortest string of the state.

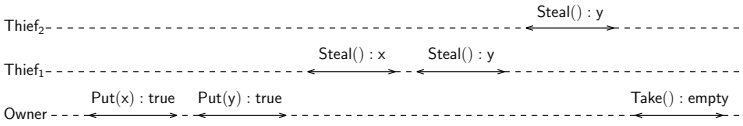


Figure 4 A seq. exec. of work-stealing with weak multiplicity.

The specification of work-stealing with concurrent multiplicity is nearly trivial, with solutions requiring almost no synchronization. A simple solution is obtained by replacing *Head* in WS-MULT with one local persistent variable *head* per process (each initialized to 1); Put remains the same and Take and Steal instead of reading from *Head*, they just locally read the current value of *head* and increment it whenever a task is taken. To avoid this kind of simple solution (which would very inefficient in practice for solving a specific problem in parallel, as essentially every process processes every task), we restrict our attention to *sequentially-exact* algorithms, namely, every sequential execution of the algorithm is a sequential execution of the specification of (FIFO) work-stealing. [2] It is easy to see that the algorithm described above does not have this property. Finally, we stress that, differently

[2] Alternatively, work-stealing with weak multiplicity can be specified using the interval-linearizability formalism in [9], which allows us to specify that a Take/Steal operation can exhibit a non-exact only in presence of concurrency. Interval-linarizability would directly imply that any interval-linearizable solution provides an exact solution in sequential executions. Roughly speaking, in interval-linearizability, operations are linearized at intervals that can overlap each other.

from work-stealing with multiplicity, two distinct non-concurrent Take/Steal can extract the same task in an execution, which can happens only if *some* operations are concurrent in the execution, due to the sequentially-exact requirement. Particularly, in our algorithms, this relaxed behaviour can occur when processes are concurrently updating the head of the queue.

Fully Read/Write Fence-Free Work-Stealing with Weak Multiplicity. We present WS-WMULT, a fully Read/Write fence-free algorithm for work-stealing with weak multiplicity. The algorithm is obtained by replacing the atomic MaxRegister object in WS-MULT, $Head$, with an atomic RangeMaxRegister object, a relaxation of MaxRegister, defined below.

We present a RangeMaxRegister algorithm that is nearly trivial, however, it allows us to solve work-stealing with weak multiplicity in an efficient manner, with implementations exhibiting good performance in practice, as we will see in Section 7. As above, to avoid trivial solutions, we focus on sequentially-exact linearizable algorithms for RangeMaxRegister, i.e. each sequential executions of the algorithm is a sequential execution of MaxRegister.

```
Shared Variables:
    Head: atomic Read/Write object initialized to 1
    Tasks[1, 2, ...]: array of atomic Read/Write objects
                with the first two objects initialized to ⊥
Persistent Local Variables of the Owner:
    head ← 1
    tail ← 0
Persistent Local Variables of a Thief:
    head ← 1

Operation Put(x):
(01) tail ← tail + 1
(02) {Tasks[tail].Write(x), Tasks[tail + 2].Write(⊥)}
(03) return true
end Put

Operation Take():
(04) head ← max{head, Head.Read()}
(05) if head ≤ tail then
(06)    {x ← Tasks[head].Read(), Head.Write(head + 1)}
(07)    head ← head + 1
(08)    return x
(09) return empty
end Take

Operation Steal():
(10) head ← max{head, Head.Read()}
(11) x ← Tasks[head].Read()
(12) if x ≠ ⊥ then
(13)    Head.Write(head + 1)
(14)    head ← head + 1
(15)    return x
(16) return empty
end Steal
```

Figure 5 WS-WMULT algorithm with the RangeMaxRegister algorithm in Figure 6 inlined.

When WS-WMULT is combined with this algorithm, it becomes fully Read/Write, fence-free, linearizable, sequentially-exact and wait-free with constant step complexity. Intuitively, in RangeMaxRegister, each process has a *private* MaxRegister and whenever it invokes RMaxRead, the result lies in the range defined by the value of its private MaxRegister and the maximum among the values of the private MaxRegisters in the state. In the sequential specification of RangeMaxRegister, each invocation/response of p_i is subscripted with i.

▶ **Definition 7** (RangeMaxRegister). *The set of states Q is the infinite set of n-vectors with natural numbers, with vector $(1,\ldots,1)$ being the initial state. $\forall(r_0,\ldots,r_{n-1}) \in Q$ and $i \in \{0,\ldots,n-1\}$, the transitions are the following:*

1. *If $x > r_i$ then $\delta((r_0,\ldots,r_{n-1}),$ $\mathsf{RMaxWrite}_i(x)) =$ $((r_0,\ldots,r_{i-1},x,r_{i+1},\ldots,r_{n-1}),$ $\langle\mathsf{RMaxWrite}_i(x) : \mathsf{true}\rangle)$, otherwise $\delta((r_0,\ldots,r_{n-1}),\mathsf{RMaxWrite}(x)) =$ $((r_0,\ldots,r_{n-1}),\langle\mathsf{RMaxWrite}(x) : \mathsf{true}\rangle)$.*
2. *$\delta((r_0,\ldots,r_{n-1}),\mathsf{RMaxRead}_i()) = \{((r_0,\ldots,r_{n-1}),\langle\mathsf{RMaxRead}_i() : x\rangle)\}$, where $x \in \{r_i, r_i+1,\ldots,\max(r_0,\ldots,r_{n-1})\}$.*

As already mentioned, WS-WMULT is the algorithm obtained by replacing *Head* in WS-MULT with an atomic RangeMaxRegister object initialized to 1 (hence MaxRead and MaxWrite are replaced by RMaxRead and RMaxWrite, respectively).

▶ **Theorem 8.** WS-WMULT *is a linearizable wait-free fence-free algorithm for work-stealing with weak multiplicity using atomic* Read/Write *objects and a single atomic* RangeMaxRegister *object. All operations have constant step complexity and* Put *is fully* Read/Write.

▶ **Theorem 9.** *The algorithm in Figure 6 is a linearizable sequentially-exact wait-free fence-free algorithm for* RangeMaxRegister *using only atomic* Read/Write *objects and with constant step complexity in all its operations.*

▶ **Theorem 10.** *If Head is an instance of the algorithm in Figure 6,* WS-WMULT *is fully* Read/Write*, fence-free, wait-free, sequentially-exact and linearizable with constant step complexity in all its operations.*

Figure 5 contains an optimized version of WS-WMULT with the RangeMaxRegister algorithm in Figure 6 inlined. Since Take and Steal first RMaxRead from and then RMaxWrite to *Tail*, the algorithm remains sequentially-exact when removing Line 1 of RMaxWrite in Figure 6. Our experimental evaluation in Section 7 tested implementations of this algorithm.

```
Shared Variables:
    R: Read/Write object init. to 1
Persistent Local Var. of a Process:
    r ← 1

Operation RMaxWrite(x):
(01) r ← max{r, R.Read()}
(02) if x > r then
(03)     {r ← x, R.Write(x)}
(04) return true
end RMaxWrite

Operation RMaxRead():
(05) r ← max{r, R.Read()}
(06) return r
end RMaxRead
```

Figure 6 A linearizable wait-free algorithm for RangeMaxRegister.

5 Bounding the Multiplicity

Here we discuss simple variants of our algorithms that bound the number of operations that can extract the same task. We only discuss the case of WS-MULT as the variants for WS-WMULT are similar.

Bounding multiplicity. The modification consists in having an extra array A of the same length of $Tasks$, with its first two entries initialized to true. Steal is modified as follows: after Line 11, a thief performs $A[head]$.Swap(false), and it executes Lines 12 and 13 only if the Swap successfully takes the true value in $A[head]$; otherwise, it goes to Line 9 to start over. The modified algorithm guarantee that no two distinct Steal operations take the same task, however, a Take and a Steal can take the same task. Note that Steal is only nonblocking in the modified algorithm. This *bounded* variant of WS-MULT is denoted B-WS-MULT. The new algorithm is a set-linearizable solution to the variant of work-stealing with multiplicity, Definition 2, where every concurrency class has at most one Take and one Steal that return the same task.

Removing multiplicity. The Take operation of B-WS-MULT can be modified similarly to obtain an algorithm for exact (FIFO) work-stealing, i.e., every task is taken *exactly* once (Definition 2 with singleton concurrency classes). The modified Take operation remains wait-free.

Multiplicity on demand. Consider a variant of Definition 2 in which a task x encodes if it can be executed by several processes, denoted mult(x), or it has to be executed by a single process, denoted $\neg$mult(x) (in practice this can be done, for example, by stealing a bit from the task representation). Then, WS-MULT can be modified to have multiplicity *on demand*. In the modified Take operation, after executing the operations in Line 6, the owner tests if mult(x) holds, and if so, it returns x; otherwise, it performs $Tasks[head]$.Swap($\top$), and then returns x only if the Swap successfully takes the task in $Tasks[head]$, else it goes to Line 4 to start over. In the modified Take operation, after Line 10, a thief checks if $x = \bot$, and if so, it returns empty. Then, it checks if $x \neq \top$ and mult(x) holds, and if so it returns x. Otherwise, $x \neq \top$ and $\neg$mult(x) holds, and the thief performs $Tasks[head]$.Swap($\top$) and returns x only if the Swap returns a value distinct from $\top$, else it goes to Line 9 to start over. In the resulting algorithm, if mult(x) holds, x is taken by one operation. The modified Take operations remains wait-free but the modified Steal operation is only nonblocking.

6 Coping with realistic assumptions

We have presented our algorithms assuming all base objects used for manipulating/implementing the head and the tail are able to store values of unbounded length. However, we can assume that these base objects can store only 64-bit values. This makes our algorithms *bounded* as at most 2^{64} tasks can be inserted, but arguably this number is enough in any real application. We also have assumed that there is an array of infinite length where tasks are stored. We now discuss two approaches to remove this assumption; both approaches have been used in previous algorithms (e.g. [1, 2, 25, 16, 30]). We only discuss the case of WS-MULT as the other cases are handled in the same way.

In the first approach, the algorithm starts with $Tasks$ pointing to an array of finite fixed length, with its two first objects initialized to $\bot$; each time the owner detects the array is full (i.e. when *tail* is larger the length $Tasks$), in the middle of a Put operation, it creates a new array A, duplicating the previous length, copies the previous content to A, initializes the next two objects to $\bot$, points $Tasks$ to A and finally continues executing the algorithm. Although the modified Put operation remains wait-free, its step complexity is unbounded.

In the second approach, $Tasks$ is implemented with a linked list with each node having a fixed length array. Initially, $Tasks$ consists of a single node, with the first two objects of its array initialized to $\bot$. When the owner detects that all entries in the linked list have been used, in the middle of a Put operation, it creates a new node, initializes the first two

objects to $\perp$, links the new node to the end of the list and continues executing the algorithm. An index of $Tasks$ is now made of a tuple: a pointer to a node of the linked list and an node-index array. Thus, any pair of nodes can be easy compared (first pointer nodes, then node-indexes) and incrementing an index can be easily performed too (if the node-index is the last one, the pointer moves forward and the node-index is set to one, otherwise only the node-index is incremented). The modified Put operation remains wait-free with constant step complexity. Our experimental evaluation in Section 7 shows the second approach performs better than the first one when solving a problem of concurrent nature.

7 Experiments

Here we discuss the outcome of the experiments we have conducted to evaluate the performance of WS-WMULT and its bounded version, B-WS-WMULT (see Section 5). Based on the approaches discussed in Section 6, we implemented two versions of the algorithms, one using arrays and another one using linked lists. We mainly discuss the results of the version based on linked lists, since it exhibited better performance than the version based on arrays. WS-WMULT and B-WS-WMULT were compared to the following algorithms: Cilk THE [12], Chase-Lev [15], and the three Idempotent Work-Stealing algorithms [25].

Platform and Environment. The experiments were executed in two machines with distinct characteristics. The first machine has an Intel Kaby Lake processor (i7–7700HQ, with four cores where each core has two threads of execution) and 16GB RAM. This machine was dedicated only to the experiments. The second one is a machine with four Intel Xeon processors (E7–8870 v3 processor) and 3TB RAM; each processor has 18 cores, with each core executing two threads, so in total it has 72 cores where 144 threads can be executed in parallel. This machine was shared with other users, where the process executions are closer to a usual situation, namely, resources (CPU and memory) are shared by all users. We will focus mainly on the results of the experiments in the Core i7 processor, and additionally will use the results of the experiments in Intel Xeon to complement our analysis. All algorithms were implemented in the Java platform (OpenJDK 1.8.0_275) in order to test them in a cross-platform computing environment.

Methodology. To analyze the performance of the algorithms, we divided the analysis into the next two benchmarks, which have been used in [15, 25, 26]: (1) *Zero cost experiments.* (2) *Irregular graph application.* Below, we explain in detail how we implement each benchmark.

Zero cost experiments. We measure the time required for performing a sequence of operations provided by the work-stealing algorithms. We measure the time needed for Put-Take operations, where the owner performs a sequence of Put operations followed by an equal number of Takes. Differently from [25, 26], we also measure the time for Put-Steal operations. In both experiments, The number of Put operations is $10,000,000$, followed by the same number of Take or Steal operations; no operation performs any work associated to a task. The idea of the zero-cost experiments is to show that the presence of heavy synchronization mechanisms of an algorithm slows down the computation even in sequential executions.

Irregular graph application. We consider the spanning-tree problem to evaluate the performance of each algorithm. It is used to measure the speed-up of the computation by the parallel exploration of the input graph. This problem is used in [25, 26] to evaluate their work-stealing algorithms. We refer the reader to [6] for a detailed description of the algorithm.

The spanning tree algorithm already uses a form of work-stealing to ensure load-balancing, and we adapted it to work with all tested work-stealing implementations. We implemented all algorithms in [15, 25, 26] in Java as the experiments in those paper were perfomed in C/C++.

Further, the algorithms were tested on many types of directed and undirected graphs. We use the following graphs: *2D Torus*, *2D60 Torus*, *3D Torus*, *3D40 Torus* and *Random*. All graphs are represented using the adjacency lists representation. Each experiment consists of the following: take one of the previous graphs, specify its parameters and run the spanning-tree algorithm with one of the work-stealing algorithms (Cilk THE, Chase-Lev, Idempotent FIFO, Idempotent LIFO, Idempotent DEQUE, WS-WMULT or B-WS-WMULT); the spanning tree algorithm is executed five times with each work-stealing algorithm, using the same root and the same graph. In each execution, we test the performance obtained by increasing the number of threads from one to the total supported by the processor and registering the duration of every execution. For each work-stealing algorithm and number of threads, the fastest and slowest executions are discarded (similarly to [25]), and the average of the remaining three is calculated. All results are normalized with respect to Chase-Lev with a single thread, as in [25, 26]. In the Appendix, we discuss about the initial length of arrays and lengths of node-arrays in the experiments, as well as the number of vertices used in the graphs.

Results. We present a summary of the results before explaining them in detail: (1) *Zero cost experiments.* WS-WMULT has a similar performance than other algorithms in the Put-Take experiment. However, WS-WMULT exhibited better performance in Put-Steal experiment. This happens because the Steal operation of any other algorithm uses costly primitives, like Compare&Swap or Swap. B-WS-WMULT has the worst performance in the Put-Steal experiment, due to the management of the additional array for marking a task as taken. However, interestingly this result does not preclude the algorithm for exhibiting a competitive performance in the next benchmark. (2) *Irregular graph application.* In general WS-WMULT has better performance than any other algorithm and in particular it performed better than Idempotent FIFO in virtually all cases. B-WS-WMULT has a lower performance than WS-WMULT but still competitive respect to Idempotent algorithms. Below we discuss in detail the results of the zero cost experiments and the irregular graph application, in both cases omitting the results of Cilk THE because its performance was similar to that of Chase-Lev. Similarly, we omit the results of Idempotent DEQUE since in general it had the worst performance among the Idempotent algorithms.

Zero cost experiments Figure 9a depicts the result of the Put-Take experiment in the Intel Core i7 processor. The results of Cilk THE and the idempotent algorithms are similar than those presented [25]. As for WS-WMULT, the time required for Put operations was similar than that of Idempotent LIFO, and slightly faster than Idempotent FIFO. Considering the whole experiment, WS-WMULT was faster than any other algorithm. The results show a significant speed-up, where the gain is between 6% and 22% respect to the other algorihtms. For the case of B-WS-WMULT, it required about twice the time of any other algorithm. This poor performance is due to the use of an extra boolean array for bounding multiplicity. Particularly, the Put operation writes three entries of an array (two entries of $Tasks$ and one of the extra boolean array), differently from WS-WMULT's Put operations that writes only two (both of $Tasks$). The results of the Put-Steal experiment in the processor Intel Core i7 are shown in Figure 9b, where, as expected, Put operations exhibited a similar performance as in the Put-Take experiment. In the case of Steal operations, we observe that Chase-Lev,

Idempotent FIFO, and WS-WMULT are faster than Idempotent LIFO and B-WS-WMULT. In particular, WS-WMULT is the fastest among all algorithms, which is expected as it does not use fences and Read-Modify-Write instructions. We observe a gain between 11% and 37% compared to other algorithms. As for total time, the speed-up was between 19% and 40%. For B-WS-WMULT, we have a similar result than the one in the Puts-Takes experiment. Finally, the array-based implementation of WS-WMULT, denoted WS_WMULT_ARRAY in the figures, performed much better than any other algorithm, with a speed-up between 49% and 66% respect to the other algorithms in the Put-Steal experiment and for the Put-Take experiment, the array-based implementation is better by 42% - 52%. This implementation performs better than the linked-list version of WS-WMULT because the latter creates a large number of arrays during the execution. In the next experiment, we do not discuss in detail the result of the array-based implementations of our algorithms as the linked-list implementations have a similar performance, and sometimes outperforming them.

Irregular graph application. Table 7 displays the minimum and maximum speed-up achieved by each algorithm, normalized with respect to the one-thread execution of Chase-Lev, for every graph and processor. Overall, WS-WMULT had a better performance with maximum speed-up of 9.64 (in random undirected graph), corresponding to a cap gain of 5% respect to the maximum speed-up achieved by the other algorithms in the Xeon processor. With respect to the other algorithms, we observe an improvement range between 2% and 20%. B-WS-WMULT always shown a similar performance than Idempotent FIFO algorithm; we expected this behavior as both use expensive Read-Modify-Write instructions in Steal. Below we discuss the results only for the 2D Torus, in the appendix we discuss all other graphs.

Graph	Graph type	Processor	Chase-Lev	FIFO	LIFO	WS-WMULT	B-WS-WMULT
Random	Directed	Core i7	$1.0x \sim 3.22x$	$0.78x \sim 3.14x$	$0.91x \sim 3.33x$	$0.78x \sim 3.18x$	$0.74x \sim 2.99x$
		Xeon	$1.0x \sim 4.49x$	$0.81x \sim 4.82x$	$0.88x \sim 4.84x$	$0.8x \sim 4.95x$	$0.79x \sim 4.87x$
	Undirected	Core i7	$1.0x \sim 3.45x$	$0.95x \sim 3.67x$	$0.98x \sim 3.75x$	$0.9x \sim 3.66x$	$0.85x \sim 3.45x$
		Xeon	$1.0x \sim 9.12x$	$0.97x \sim 9.39x$	$1.11x \sim 9.36x$	$0.92x \sim 9.64x$	$0.94x \sim 9.57x$
2D Torus	Directed	Core i7	$1.0x \sim 2.11x$	$0.94x \sim 3.49x$	$0.9x \sim 2.73x$	$0.92x \sim 3.52x$	$0.85x \sim 3.34x$
		Xeon	$0.74x \sim 2.63x$	$1.21x \sim 5.19x$	$0.9x \sim 5.42x$	$1.32x \sim 5.33x$	$1.26x \sim 5.23x$
	Undirected	Core i7	$1.0x \sim 1.94x$	$0.61x \sim 2.14x$	$0.84x \sim 1.96x$	$0.62x \sim 2.2x$	$0.56x \sim 2.06x$
		Xeon	$1.0x \sim 1.98x$	$0.43x \sim 2.2x$	$0.77x \sim 2.33x$	$0.43x \sim 2.25x$	$0.35x \sim 2.14x$
2D60 Torus	Directed	Core i7	$1.0x \sim 2.06x$	$0.59x \sim 2.21x$	$0.87x \sim 2.15x$	$0.6x \sim 2.32x$	$0.56x \sim 2.13x$
		Xeon	$1.0x \sim 1.95x$	$0.66x \sim 2.38x$	$0.86x \sim 2.09x$	$0.64x \sim 2.39x$	$0.6x \sim 2.43x$
	Undirected	Core i7	$1.0x \sim 2.19x$	$0.77x \sim 2.71x$	$0.88x \sim 2.55x$	$0.78x \sim 2.81x$	$0.73x \sim 2.51x$
		Xeon	$1.0x \sim 2.57x$	$0.56x \sim 3.66x$	$0.8x \sim 3.29x$	$0.57x \sim 3.79x$	$0.56x \sim 3.61x$
3D Torus	Directed	Core i7	$1.0x \sim 2.21x$	$0.92x \sim 3.17x$	$0.88x \sim 2.79x$	$0.91x \sim 3.47x$	$0.86x \sim 3.26x$
		Xeon	$0.7x \sim 3.52x$	$0.92x \sim 5.75x$	$1.02x \sim 4.64x$	$1.01x \sim 5.75x$	$0.94x \sim 5.65x$
	Undirected	Core i7	$1.0x \sim 2.5x$	$0.73x \sim 2.51x$	$0.9x \sim 2.56x$	$0.67x \sim 2.58x$	$0.62x \sim 2.52x$
		Xeon	$1.0x \sim 3.77x$	$0.83x \sim 4.94x$	$0.94x \sim 3.94x$	$0.85x \sim 4.87x$	$0.76x \sim 5.05x$
3D40 Torus	Directed	Core i7	$1.0x \sim 2.95x$	$0.88x \sim 3.19x$	$0.91x \sim 3.06x$	$0.86x \sim 3.26x$	$0.82x \sim 2.99x$
		Xeon	$1.0x \sim 3.63x$	$0.8x \sim 4.46x$	$0.88x \sim 3.99x$	$0.8x \sim 4.47x$	$0.8x \sim 4.43x$
	Undirected	Core i7	$1.0x \sim 2.65x$	$0.8x \sim 3.05x$	$0.89x \sim 2.84x$	$0.78x \sim 3.02x$	$0.72x \sim 2.78x$
		Xeon	$1.0x \sim 3.67x$	$0.85x \sim 4.78x$	$0.96x \sim 3.95x$	$0.79x \sim 4.98x$	$0.85x \sim 4.81x$

Figure 7 Summary of speedups of each algorithm. Only maximum and minimum are shown.

2D Torus. In the case of 2D Torus (and similarly for the 2D60 Torus), we observe a pattern: our algorithms showed a bad performance with one or two threads, but when the number of threads increases, our algorithms improved the speed up of Idempotent LIFO and Chase-Lev. Additionally, the performance showed by Idempotent FIFO is between the registered by WS-WMULT and B-WS-WMULT. We can see this behavior in the Figure 10a. The results show the following: for the 2D Directed Torus, the gains range of WS-WMULT is from 1% to 40% with respect other algorithms. In a similar comparison, for the 2D Undirected Torus, the gains range of WS-WMULT is of 3% to 12% and for the 2D60 Undirected Torus

and its Directed version, the gains are between 5% to 11% and 4% to 22% respectively. This behavior is consistent in both machines. In most of the of executions, WS-WMULT has a better performance than other algorithms (see Figure 10b).

Final remark. In our parallel spanning-tree experiments, only a small fraction of the tasks are repeated. One reason for this to happen is that methods of WS-WMULT are short and simple, and hence very often a process has enough time in a core to grab a task only for itself (a task can be taken several times olnly in specific concurrent scenarios). In other words, since the algorithm is very simple, an optimistic approach gives exclusive access to a task, most of the time.

8 Final Discussion

We have studied two relaxations for work-stealing, called multiplicity and weak multiplicity. Both of them allow a task to be extracted by more than one Take/Steal operation but each process can take the same task at most once; however, the relaxation can arise only in presence of concurrency. We presented fully Read/Write wait-free algorithms for the relaxations. All algorithms are devoid of Read-After-Write synchronization patterns and the algorithm for the weak multiplicity case is also fully fence-free with constant step complexity. To our knowledge, this is the first time work-stealing algorithms with these properties have been proposed, evading the impossibility result in [4] in all operations. From the theoretical perspective of the consensus number hierarchy, we have shown that work-stealing with multiplicity and weak multiplicity lays at the lowest level of the hierarchy [19]. We also argued that the idempotent work-stealing [25] do not solve work-stealing with multiplicity (see Section 7 in the Appendix), therefore the relaxations and algorithms proposed here provide stronger guarantees. We also have performed an experimental evaluation comparing our work-stealing solutions to Cilk THE, Chase-Lev and idempotent work-stealing algorithms. The experiments show that our WS-WMULT algorithm exhibits a better performance than the previously mentioned algorithms, while its bounded version, B-WS-WMULT, with a Swap-based Steal operation, shows a lower performance than WS-WMULT, but keeps a competitive performance with respect to the other algorithms. All our results together show that one of the simplest synchronization mechanisms suffices to solve non-trivial coordination problems, particularly, a relaxation of work-stealing that helps with computing a spanning tree.

For future research, we are interested in developing algorithms for work-stealing with multiplicity and weak multiplicity that insert/extract tasks in orders different from FIFO. Also, it is interesting to explore if the techniques in our algorithms can be applied to efficiently solve relaxed versions of other concurrent objects. For example, it is worthwhile to explore if a multi-enqueuer multi-dequeuer queue with multiplicity (resp. weak multiplicity) can be obtained by manipulating the tail with a MaxRegister (resp. RangeMaxRegister) object, like the head is manipulated in our algorithms.

References

1 Dolev Adas and Roy Friedman. Brief announcement: Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 50:1–50:3, 2020. doi:10.4230/LIPIcs.DISC.2020.50.

2 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010. doi:10.1007/978-3-642-17653-1_29.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. doi:10.1145/2108242.2108244.

4 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498, 2011. doi:10.1145/1926385.1926442.

5 Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009. doi:10.1109/TPDS.2008.105.

6 David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004. doi:10.1109/IPDPS.2004.1302951.

7 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995*, pages 207–216, 1995. doi:10.1145/209936.209958.

8 Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity. *CoRR*, abs/2008.04424, 2020. arXiv:2008.04424.

9 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

10 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, pages 13:1–13:19, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.

11 Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538, 2005. doi:10.1145/1094811.1094852.

12 David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 21–28, 2005. doi:10.1145/1073970.1073974.

13 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

14 Christine H. Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, 2001. URL: http://www.usenix.org/publications/library/proceedings/jvm01/full_papers/flood/flood.pdf.

15 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223, 1998. doi:10.1145/277650.277725.

16 Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Comput.*, 18(3):189–207, 2006. doi:10.1007/s00446-005-0144-5.

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. doi:10.1145/571825.571876.

18 Maurice Herlihy. Impossibility results for asynchronous PRAM (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, July 21-24, 1991*, pages 327–336, 1991. doi:10.1145/113379.113409.

19 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

20 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

21 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

22 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, pages 408–419, 2005. doi:10.1007/11590156_33.

23 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000. doi:10.1137/S0097539797317299.

24 Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, pages 36–43, 2000. doi:10.1145/337449.337465.

25 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. doi:10.1145/1504176.1504186.

26 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. doi:10.1145/2541940.2541987.

27 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396, 1994. doi:10.1145/197917.198176.

28 Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 13–24, 2007. doi:10.1109/HPCA.2007.346181.

29 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.

30 Chaoran Yang and John M. Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 16:1–16:13, 2016. doi:10.1145/2851141.2851168.

A Idempotent ≠ Multiplicity

Idempotent work-stealing is (only) informally defined in [25] as: every task is extracted *at least once*, instead of *exactly once* (in some order). Here we explain that these algorithms does not implement work-stealing with multiplicity, neither its non-concurrent variant. While in our relaxations every process extracts a task at most once, and hence the number of

distinct operations that extract the same task is at most the number of processes in the system, in idempotent work-stealing a task can be extracted an unbounded number of times, and moreover, a thief can extract the same task an unbounded number of times.

```
   Structures:
     Task: task information
     TaskArrayWithSize:
       size: integer
       array: array of Task
     FifoIwsq:
       head: integer;
       tail: integer;
       tasks: TaskArrayWithSize

   constructor FifoIwsq(integer size) {
     head := 0;
     tail := 0;
     tasks := new TaskArrayWithSize(size);
   }

   void put(Task task) {
     Order write at 4 before write at 5
1:   h := head;
2:   t := tail;
3:   if (t = h+tasks.size) {expand(); goto 1;}
4:   tasks.array[t%tasks.size] := task;
5:   tail := t+1;
   }

   TaskInfo take() {
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   task := tasks.array[h%tasks.size];
5:   head := h+1;
6:   return task;
   }

   TaskInfo steal() {
     Order read in 1 before read in 2
     Order read in 1 before read in 4
     Order read in 5 before CAS in 6
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   a := tasks;
5:   task := a.array[h%a.size];
6:   if !CAS(head,h,h+1) goto 1;
7:   return task;
   }
```

Figure 8 Idempotent FIFO work-stealing [25].

Figure 8 depicts the FIFO idempotent work-stealing algorithm in [25]. For every integer $z > 0$, we describe an execution of the algorithm in which, for every $k \in \{1, \ldots, z\}$, there is a task that is extracted by $\Theta(k)$ distinct operations (possibly by the same thief), with only one of them being concurrent with the others.

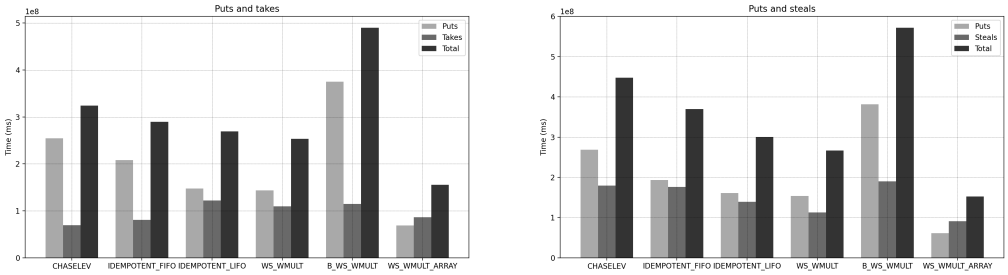
1. Let the owner execute alone z times Put. Thus, there are z distinct tasks in $tasks$
2. Let $r = z$.
3. The owner executes Take and stops before executing Line 5, i.e. it is about to increment $head$.
4. In some order, the thieves sequentially execute r Steal operations; note these Steal operations return the r tasks in $tasks[0, \ldots, r-1]$.
5. We now let the owner increment $head$. If $r > 1$, go to step 3 with r decremented by one, else, end the execution.

Observe that in the execution just described, the task in $tasks[i]$, $i \in \{0, \ldots, z-1\}$, is extracted by a Take operation and by $i+1$ distinct non-concurrent Steal operations (possible by the same thief). Thus, the task is extracted $\Theta(i)$ distinct times. Since z is any positive integer, we conclude that there is no bound on the number of times a task can be extracted.

A similar argument works for the other two idempotent work-stealing algorithms in [25]. In the end, this happens in all algorithms because tasks are not marked as taken in the shared array where they are stored. Thus, when the owner takes a task and experience a delay before updating the head/tail, all concurrent modifications of the head/tail performed

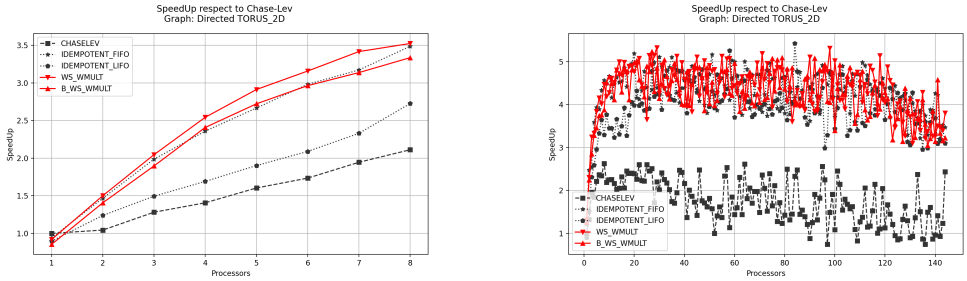
by the thieves are overwritten once the owner completes its operation, hence leaving all taken tasks ready to be taken again.

B Additional Figures



(a) Results of experiment for puts and takes. **(b)** Results of experiment for puts and steals.

Figure 9 Zero cost experiments, less time is better.



(a) Speed up on Intel Core i7. **(b)** Speed up on Intel Xeon.

Figure 10 Speed-ups of algorithms using directed torus 2D graph.