Efficient Synchronization-Light Work Stealing

Rafael Custódio
r.custodio@campus.fct.unl.pt
Department of Computer Science,
NOVA School of Science and
Technology, NOVA University Lisbon
Portugal

Hervé Paulino
herve.paulino@fct.unl.pt
Department of Computer Science &
NOVA LINCS, NOVA School of
Science and Technology, NOVA
University Lisbon
Portugal

Guilherme Rito
guilherme.teixeira@inf.ethz.ch
Department of Computer Science,
ETH Zurich
Switzerland

ABSTRACT

Work Stealing (WS) is a provably efficient scheduler of parallel computations. In WS each processor owns a deque that it uses as a call stack; when out of work, processors try to steal tasks from other processors’ deques. Unfortunately, the concurrent nature of processors’ deques entails expensive synchronization even when processors access their own deques. Recently, Rito and Paulino have found that the use of split deques allows to provably avoid most synchronization costs while keeping WS’s asymptotically optimal expected runtime [27]; in Low-Cost Work Stealing (LCWS)—the variant of WS introduced in their work—processors need not synchronize for most local accesses to their (split) deques.

In this paper we assess the concrete efficiency gains of LCWS in practice. More concretely, we implemented LCWS in the Parlay library and show how it compares against Parlay’s original work stealing algorithm on the execution of the benchmarks from the Problem-Based Benchmark Suite (PBBS). Experimental results show that our signal-based LCWS implementation obtains speedups with regard to WS for at least 65% of PBBS’ benchmarks in three different computers.

CCS CONCEPTS

• Computing methodologies → Concurrent algorithms; Shared memory algorithms; Parallel computing methodologies.

KEYWORDS

scheduling; load balancing; work stealing; synchronization-light; runtime systems

ACM Reference Format:

1 INTRODUCTION

1.1 Motivation

Parallel programming is key to take full advantage of modern computer chips’ processing capabilities. In this context, numerous parallel programming aids have been developed, from programming libraries like the Java concurrency API [14] and Intel’s TBB [20], among others [7, 8, 17, 19], to extensions of programming languages, such as OpenMP [6], natively supported constructs, such as C++’s async [18] and Go’s goroutines [13], and even entirely new programming languages like Cilk++ [21] and X10 [15]. A primary objective of these programming tools is to provide developers with a straightforward method for specifying how their program’s execution can be divided into smaller units called tasks, which can be run concurrently. Once this information is provided, these tools then determine the scheduling of the program’s execution.

Work Stealing (WS) is a provably efficient scheduler of parallel computations: the expected runtime of WS for executing a computation with total work (i.e. total number of instructions) $W$ and span (i.e. critical-path length) $S$ on $P$ processors (or workers) is $O(W/P + S)$, which is a constant factor away from optimal [4, 9]. WS’s efficiency is not just theoretical: it is the de facto standard scheduler used in the runtime systems of parallel programming. In fact, as shown by Attiya et al. in [5], expensive synchronization operations are necessary even for the deque’s owner when it is operating locally, only accessing the bottom of its deque.

In recent work, Rito and Paulino have shown that replacing WS’s fully concurrent deques with split deques allows to (provably) eliminate most synchronization operations when processors are operating locally on their deques [27]. More concretely, they prove that their scheduling algorithm—Low-Cost Work Stealing (LCWS)—not only maintains WS’s asymptotically optimal expected runtime but also provably avoids most synchronization operations: for a computation with total work $W$ and span $S$ the total expected time that processors spend executing synchronization operations in a
\( P \)-processor execution using LCWS is upper bounded (roughly) by \( O(SP) \).

**Multiprogrammed Environments.** In standard computing environments like desktop and mobile devices, the aforementioned parallel programming aids are responsible for scheduling a program’s computation through user-level threads (to which we refer to as processors). However, these tools usually allocate resources without considering the global load of the system. As a result, when multiple runtime systems coexist in a computing system, they compete for available processing power instead of cooperating, leading to interference and reduced overall performance.

To address this issue, various works have focused on dynamic assignment and reallocation of hardware resources (such as processing cores, caches, and memory bandwidth) either among runtime systems [11, 16, 22, 23], among jobs within an runtime system [32] or at operating system level [25]. This means that, on one hand, when a parallel computation has access to all available computing resources, its runtime scheduler can fully utilize these resources for efficient execution. On the other hand, even when only a fraction of the computing resources are available, the runtime scheduler should still effectively utilize them. We, however, observe that it not the case: the impact of WS’s synchronization costs intensify when the number of processors is low, given that most tasks are executed by the processor that generates them and, thus, synchronization should not have been necessary.

In this paper we address this problem by using (an implementation of) the LCWS scheduler [27] to efficiently load-balance computations in scenarios with both high and low number of processors. Our goal is to ensure a high efficiency of the runtime system independently of how much resources are used by the runtime system: when many processors are used we aim at performances on par with WS, and in the case of few processors being used we aim at performances surpassing WS.

### 1.2 Contributions

In this paper we study the performance of LCWS in practice, comparing it to the standard WS scheduler. We evaluate these implementations in the context of the Parlay parallel processing toolkit [8], applying it to all input instances of all benchmarks that compose the Problem-Based Benchmark Suite (PBBS) Version 2 [3]. Our contributions are:

1. A set of work stealing-based schedulers that embody the concepts presented in LCWS. We present and discuss the rationale behind several implementations:
   - User space only implementation of LCWS (Sections 3 and 3.2). This is mostly an approximation to a concrete C++ implementation of LCWS [27] which requires no intervention of the operating system. As will be discussed ahead, this implementation does not correspond to a provably efficient version of LCWS, and suffers from issues similar to Lace [31].
   - Signal-based versions (Sections 4, 4.1.1 and 4.1.2) that make use of signaling among threads to improve efficiency when the number of processors is high. We experiment

2. The integration of the schedulers in the Parlay toolkit, making them available for the community to use: the code can be accessed from https://bitbucket.org/marrow-project/lcws/. Any software system build atop Parlay can benefit from the properties of the proposed schedulers without changing the system’s source code.

3. A comprehensive evaluation (Section 5) that compares our schedulers against the (default, unchanged) WS scheduler of Parlay. Our evaluation is based on PBBS Version 2, and in particular uses all input instances of all benchmarks that compose PBBS. Experimental results show that our signal-based LCWS implementation obtains speedups with regard to WS of 65% to 69% of all benchmarks configurations executed in three different computers.

### 2 RELATED WORK

**The LCWS Algorithm** [27]. In traditional WS schedulers processors need to execute expensive synchronization instructions when accessing deques [5] to guarantee correctness. While this is necessary in standard WS schedulers due to the fully concurrent nature of its deques—wherein every item/task in the deque can be taken from any processor at any time—recent work by Rito and Paulino [27] has shown it is possible to avoid most of these synchronization overheads by using split-deques—a concurrent deque hand-tuned for WS that by default keeps tasks local to the owner (thus not requiring synchronization when accessing such local items), but still allowing tasks to be taken by other processors (when the deque’s owner chooses to share them, allowing for load balancing). For a comprehensive discussion of approaches to avoid synchronization costs we refer the reader to [27]; below, we focus only on implementations of WS algorithms that aim at reducing synchronization by making part (or the entirety) of deques private to their owners.

**A Note on Signaling and the LCWS Algorithm** [27]. Instead of using signals, the original LCWS scheduler relies on a custom notification mechanism embedded into the scheduler itself [27]. As stated in [27, First paragraph of Section 3.2], the reason behind this is that the goal of their work is to bound the synchronization costs of their algorithm (LCWS) and so it is crucial to make all possible sources of synchronization explicit. Nevertheless, in [27, First Paragraph of Section 3.2 and Point 2 of Section 1.1] it is noted that LCWS’s notification mechanism could in practice be implemented using signals: as long as work exposure requests are attended in constant time, the expected runtime of LCWS is asymptotically optimal.

**Lace** [31]. Dijk and Pol propose a variant of WS—one which is rather similar to LCWS—wherein split deques are also used in place of the typical fully concurrent deques [31]. Similarly to LCWS, their scheduler also relies on thieves requesting their victims to expose work via a flag. However, in sharp contrast to LCWS (and to the signal-based LCWS implementations we give in this paper), their scheduler does not handle work exposure requests in constant time: busy processors only check if they have been requested work when they access their deque. We note that our user space implementation of LCWS suffers from a similar issue (see Section 3). To understand...
When load balancing, thieves communicate (synchronously) with a task when it is exposed—i.e. when a processor transfers the task to the public part of its split deque (synchronization-free) call stack; the public part is also accessible by thieves, and is used for load balancing. As in Work Stealing (WS), when a processor is out of work it tries to steal tasks from other processors’ split deques. However, and in contrast to WS, thieves cannot always steal tasks directly from their victims split deques: they can only steal the ones that are in the public part of a victim’s split deque. To ensure proper load balancing while keeping the private part of processors’ split deques synchronization-free, LCWS relies on an asynchronous notification mechanism that thieves use to inform their victims they were targeted (for work stealing). Once notified, a victim (i.e. the targeted processor) transfers the topmost task of the private part of its deque (if any) to the

Work Exposure Strategies of LCWS [27] and Lace [31]. In LCWS, thieves can request work whenever a processor transfers the task to the public part of its split deque, making it available for thieves to steal—it is never “unexposed”—i.e. transferred to the private part of the split deque’s owner, so it cannot be taken by thieves. This is in contrast to Lace [31], where it is possible that a task that was exposed is “unexposed”; this can occur if the owner of a split deque realizes that the private part of its split deque is empty but the public part (where the tasks are exposed to thieves) is not.

WS with Private Deques [2]. Acar et al. propose to avoid synchronization by making deques entirely private to their owners [2]: when load balancing, thieves communicate (synchronously) with victims to request work. This in particular means that even when processors are busy (working locally on their deques), they still need to check for incoming work requests by thieves. Although Acar et al. noted that one could resort to interrupts in order to make the polling more efficient (see [2, Section 4, Handling of large sequential tasks]), this is not how their scheduler is implemented: instead, to ensure load balancing requests are handled in a timely manner their scheduler relies on an additional processor that periodically interrupts busy processors to make sure they handle incoming work requests (if any).

Fence-Free WS for Bounded TSO Processors [24]. Taking the architecture of modern TSO chips into account, Morrison and Afek avoid the synchronization overheads of WS schedulers by keeping part of processors deques entirely private [24]; the size of the part of deques that is kept private is calculated based on the specifications of the underlying physical microprocessors used. While this approach allows to fully eliminate synchronization overheads for local deque accesses while maintaining correctness, since the bottommost items in processors’ deques are not available for other processors to take, load balancing is limited.

Lazy Binary-Splitting WS. Tzannes et al. propose a WS scheduler where processors keep all their work entirely private except for the topmost task, which is stored in a shared cell [30]. While ensuring the topmost task can always be taken by other processors is key to proper load balancing, their approach has limitations for computations where processors may need to access the topmost nodes of their deques regularly. As mentioned in [2], a similar limitation has been identified in Chase-Lev concurrent deques [10].

Automatic Granularity Control. The concept of Automatic Granularity Control (AGC) is also close to our work, given that it can be used to reduce synchronization overheads when the processor count is low. AGC requires either: (a) rewriting the code to provide both parallel and sequential versions of it, as well as cost functions, such as in [1], or (b) sophisticated analysis to create versions of the tasks with different granularities, such as in [29]. The former requires changing the source code, while our approach does not; all benchmarks of PBBS used in Section 5 ran unmodified. The shortcomings of the latter include knowing the number of (and which) versions to generate, how to do it when in the presence of calls to external libraries and the overhead imposed by the runtime system to choose the best version to use at any given time.

3 USER SPACE IMPLEMENTATION

In Low-Cost Work Stealing (LCWS) each processor owns a (lock-free) split deque [27]. As illustrated in Figure 1, a split deque is divided into a public part and a private part: the private part is only accessible to the deque’s owner, who uses it as a regular (synchronization-free) call stack; the public part is also accessible by thieves, and is used for load balancing. As in Work Stealing (WS), when a processor is out of work it tries to steal tasks from other processors’ split deques. However, and in contrast to WS, thieves cannot always steal tasks directly from their victims split deques: they can only steal the ones that are in the public part of a victim’s split deque. To ensure proper load balancing while keeping the private part of processors’ split deques synchronization-free, LCWS relies on an asynchronous notification mechanism that thieves use to inform their victims they were targeted (for work stealing). Once notified, a victim (i.e. the targeted processor) transfers the topmost task of the private part of its deque (if any) to the

Figure 1: The split deque built from an array of elements. top points to the deque’s top-most element, public_bot points to node below the bottom-most element of the deque’s public part, and bot points to the empty slot below the deque’s bottom-most element. The processor operates on the synchronization-free part of the deque, while the thieves steal work from the public part.
3.1 Scheduler

Listing 1 presents a user space C++ implementation of the LCWS algorithm (USLCWS). As in the original algorithm, each processor owns a flag, targeted, that is used to implement a simple notification mechanism: the flag indicates if a thief targeted the flag’s owner for work stealing since the last scheduling round. However, contrary to LCWS, this notification is provided at the task- rather than instruction-level. Consequently, USLCWS does not ensure that work exposure requests are processed in constant time, and thus, does not guarantee the synchronization bounds of LCWS. However, we have chosen to implement it in order to investigate whether, in that case, the flag is also empty the processor resets its targeted flag to false (line 11), and proceeds to find another victim. If the steal attempt succeeds, the thief simply starts executing the stolen task.

3.2 Split Deque

Listing 2 presents a concrete implementation of the split deque proposed in [27]. A split deque comprises a (memory aligned) array of tasks (deq), the index below the bottom-most task in the deque (bot), the two-field structure age (comprising the top of the deque and a tag necessary to avoid the ABA problem [12]), and additionally a field public_bot that is used to keep track of where the current split is (i.e. where, currently, the private part of the split deque ends and the public part begins). We note that, contrary to the standard concurrent deque implementations for WS, neither push_bottom nor pop_bottom need any synchronization instructions (see [27, Lemmas 1 and 2]).

public_part - making such task stealable by a future thief that targets this victim.

3.1 Scheduler

Listing 1 presents a user space C++ implementation of the LCWS algorithm (USLCWS). As in the original algorithm, each processor owns a flag, targeted, that is used to implement a simple notification mechanism: the flag indicates if a thief targeted the flag’s owner for work stealing since the last scheduling round. However, contrary to LCWS, this notification is provided at the task- rather than instruction-level. Consequently, USLCWS does not ensure that work exposure requests are processed in constant time, and thus, does not guarantee the synchronization bounds of LCWS. However, we have chosen to implement it in order to investigate whether, in real-world scenarios, it is advantageous to sacrifice the theoretical bounds in favor of an implementation that operates entirely in user-space (i.e., without relying on intervention from the operating system).

In USLCWS, when looking for work, a processor first tries to obtain a task from the private part of its split deque: if successful the processor then checks if it should transfer work to the deque’s public part (via the update_public_bottom method), resets its targeted flag (lines 8 to 11), and returns the task; if the private part of the processor’s split deque is empty, it searches for work in the public part (line 15, as illustrated in Figure 2). If the public part is also empty the processor resets its targeted flag and starts a stealing phase. Otherwise, the processor removes the bottom-most task in the public part of its split deque and starts executing it.

The stealing phase is similar to the one of the WS algorithm: thieves pick their victims uniformly at random (line 20) and invoke the pop_top method to try stealing a task from the public part of the victim split deques (line 21). If the steal attempt fails (pop_top returns PRIVATE WORK), the thief sets the victim’s targeted flag to true (line 22), and proceeds to find another victim. If the steal attempt succeeds, the thief simply starts executing the stolen task.

3.2 Split Deque

Listing 2 presents a concrete implementation of the split deque proposed in [27]. A split deque comprises a (memory aligned) array of tasks (deq), the index below the bottom-most task in the deque (bot), the two-field structure age (comprising the top of the deque and a tag necessary to avoid the ABA problem [12]), and additionally a field public_bot that is used to keep track of where the current split is (i.e. where, currently, the private part of the split deque ends and the public part begins). We note that, contrary to the standard concurrent deque implementations for WS, neither push_bottom nor pop_bottom need any synchronization instructions (see [27, Lemmas 1 and 2]).

public_part - making such task stealable by a future thief that targets this victim.

Figure 2: If all work is public, the processor has to compete with the thieves to retrieve work from the deque’s public part. For that purpose, it must use method pop_public_bottom.
Listing 2: A C++ split deque implementation

template<typename Task>
struct deque {
    unsigned int public_bot, bot;
    atomic<age_t> age;
    array<aligned_task_t, size> deq;

    void push_bottom(Task* task) { deq[bot++].task = task; }

    Task* pop_bottom() {
        return bot == public_bot ? nullptr : deq[--bot].task;
    }

    Task* pop_public_bottom() {
        if (public_bot == 0) return nullptr;
        public_bot--;
        atomic_thread_fence(memory_order_seq_cst);
        auto task = deq[public_bot].task;
        auto old_age = age.load(memory_order_relaxed);
        if (public_bot > old_age.top) {
            bot = public_bot;
            return task;
        }
        bot = 0;
        auto new_age = old_age;
        new_age.top = new_age.top + 1;
        if (age.compare_exchange_strong(old_age, new_age, memory_order_relaxed)) {
            age.store(new_age, memory_order_relaxed);
            task = nullptr;
        }
        atomic_thread_fence(memory_order_seq_cst);
        return task;
    }

    Task* pop_top() {
        auto old_age = age.load(memory_order_relaxed);
        if (public_bot > old_age.top) {
            auto task = deq[old_age.top].task;
            auto new_age = old_age;
            new_age.top = new_age.top + 1;
            if (age.compare_exchange_strong(old_age, new_age, memory_order_relaxed)) {
                return task;
                return ABOFT; // set to nullptr
            } else return (public_bot < bot) ? nullptr : PRIVATE_WORK;
        }
        void update_public_bottom() {
            if (public_bot < bot) public_bot++;
        }
    }

1 2 4 8 16 32 64
0.6
0.4
0.2
0.0
0
0.2
0.4
0.6
(a) USLCWS memory fences/WS memory fences

10
20
30
40
50
60
70
80
90
100
2 4 8 16 32 64
(b) USLCWS CAS/WS CAS

20
40
60
80
100
2 4 8 16 32 64
(c) successful steals USLCWS/Successful steals WS

20
40
60
80
100
2 4 8 16 32 64
(d) Percentage of exposed work that is not stolen in USLCWS

Figure 3: Profile of USLCWS varying the number of processors. Each box reports the values measured for all benchmarks of PBBS on machine AMD32 (see Table 1)

3.3 Preliminary Evaluation of User Space LCWS

We made a preliminary evaluation of our user space implementation of LCWS. For that purpose, we followed the methodology defined for the experimental evaluation of Section 5, restricted to machine AMD32 (see Table 1). Therefore, the box plots depict the results obtained for all input instances of all benchmarks featured in Problem-Based Benchmark Suite (PBBS), run with 2 to 64 worker threads.

The experimental results, depicted in Figures 3a and 3b, confirm our expectation that LCWS greatly reduces the number of memory fences and CAS operations and when compared with WS. Note, in particular, that USLCWS uses less than 1% of memory fences and less 40% of CAS operations than WS. This is due to all local operations in USLCWS being synchronization free.
handle of the processor (target) to notify (line 8). To avoid potentially harmful compiler optimizations, the deque’s state modified in the signal handling function—namely the public_bot field—is declared volatile\(^3\).

While not needed, we have retained the targeted flag with an updated purpose, which is to prevent unnecessary system calls to pthread_kill. This user space mechanism is much more efficient than resorting to signal masks. Hence, the flag continues to be set whenever a notification (signal) is sent. However, contrarily to our user space implementation, it is only reset to false when a task is removed from the deque’s public part or the target processor pushes a new task to the deque. Ergo, if there is no work to share or the one previously shared is still accessible, no new notifications are dispatched.

A Subtlety in the Signal-Based Implementation. The use of signals may give rise to data races when a processor \(P_1\) executes method pop_bot (Listing 2) to retrieve work from the private part of its deque and concurrently a thief \(P_2\) signals \(P_1\) to expose work. Assume that the deque has a single task and that \(P_1\), in the process of executing pop_bot, has already evaluated condition \(\text{bot} \equiv \text{public_bot} \rightarrow \text{true}\). The use of signals may give rise to data races when a processor \(P_1\) executes method pop_bot (Listing 2) to retrieve work from the private part of its deque and concurrently a thief \(P_2\) signals \(P_1\) to expose work. Assume that the deque has a single task and that \(P_1\), in the process of executing pop_bot, has already evaluated condition \(\text{bot} \equiv \text{public_bot} \rightarrow \text{true}\), when the notification arrives. Given that the value of bot was not yet updated in the execution of pop_bot, the signal handler modifies the value of public_bot making the task public (i.e., exposed for thieves to take). When \(P_1\) resumes the execution of pop_bot it assumes the task is still private, and accesses the deque without the needed synchronization.

A solution that keeps method pop_bot synchronization-free is to decrement field bot prior to the comparison with public_bot and change the comparison itself to an inequality:

```c
Task* pop_bot() {
    return --bot < public_bot ? nullptr : deq[bot].task; }
```

If the comparison’s condition evaluates to false, the method’s side-effects are the same of the original version, but otherwise bot’s decrement should not occur. However, if pop_bot returns null (lines 7 to 15 of Listing 1), pop_public_bot is invoked called and thus bot is reset public_bot \(!= 0\) (see lines 15 and 18 of Listing 2). To ensure the correctness in the remainder cases we slightly modify method pop_public_bot to reset bot to 0 when public_bot is 0.

To sum up, we end up with a correct version that keeps method pop_bot synchronization-free for a rather small price: an extra decrement whenever there is no private work and, additionally, an extra assignment whenever there is also no public work.

4 SIGNAL-BASED IMPLEMENTATION

In this section, we present a truthful implementation of LCWS that provides the bounds formally proved for the scheduler and avoids the issues of the user space implementation. To that end, we resort to signaling among threads/processors, which will allow processors to handle work exposure requests in constant time\(^2\). As depicted in Listing 3, the scheduler’s state now includes a thread local variable with a thread’s scheduling identifier, as well as pthread handles of all threads. The former is needed in the signal handler to identify the thread that must transfer work to the public part of its deque (lines 3 to 5), and the latter is needed to obtain the pthread

\(^2\)Up to the time that the underlying Operating System takes to deliver signals.

\(^3\)Note that public_bot being declared volatile does not imply the execution of any synchronization operation when accessing it.
from the base implementation in that a processor only reveals work when it possesses a minimum of two tasks in the private part of its split deque. This means that the variant keeps the original implementation of pop_bottom, the update_public_bottom method only updates field public_bot if public_bot + 1 < bot, and the notification condition of (Listing 3) becomes:

```c
if (task == PUBLIC_EMPTY && !targeted[target] &&
     deques[target].has_two_tasks())
```

As its name implies, method has_two_tasks returns true if and only if the deque has at least two tasks.

4.1.2 Expose Half. In our second variant of LCWS’s signal-based implementation, when a processor is requested by a thief to expose work, it exposes half of the tasks in the private part of its split deque. We note that in this Expose Half variant, thieves still can only steal one task at a time.

Let \( r \) be the number of tasks in the private part of a processor’s split deque. To implement the Expose Half variant, we modified the update_public_bottom method making it so that if \( r \) is at least 3, the worker exposes \( \lfloor r/2 \rfloor \) tasks (and otherwise only exposes at most one task).

Although this work exposure policy is similar to Lace’s, in our implementation a processor does not “unexpose” previously exposed tasks [31]. Another important difference is, as noted in Section 2, that in our implementation, work exposure requests are handled in constant time.

**Implementation Details.** We initially used the round function provided by the C++ standard library [26]. However, this led to a significant increase—by an order of magnitude—in variant’s the runtime, when compared with the base implementation. An alternative could be using integer division, but this is also known to be slow. Aiming for a computationally lighter solution, our choice fell on a function inspired in lua_number2int32—a function featured in the Lua language’s library [28]:

```c
int double2int(double r) {
    r = 675329441855744.0;
    return reinterpret_cast<int>(r);
}
```

5 EVALUATION

To compare the performance of our schedulers against a state of the art implementation of the Work Stealing (WS) algorithm we used the (default, unaltered) Parlay parallel programming library as baseline, which relies on a well-tuned implementation of WS for load balancing [8]. Having set the baseline, we then implemented our algorithms on Parlay—replacing its WS scheduler and deque implementations by our own.

For our evaluation we resorted to the standard (unchanged) Problem-Based Benchmark Suite (PBBS) Version 2 [3], which features algorithms for graph processing, text processing, computational geometry, among others. For each benchmark the suite includes multiple input instances, each defining different workloads [3]. As an example, for the Integer Sort benchmark PBBS includes the following input instances: 1. randomSeq_100M_int, 2. exptSeq_100M_int, 3. randomSeq_100M_int_pair_int and lastly 4. randomSeq_100M_256_int_pair_int. We define a benchmark configuration as a triple

\[
(benchmark, input_instance, number\_of\_processors)
\]

where number_of_processors ranges from 1 up to the number of cores of each of the machines used in the experiments. The experimental results given in this section (and also Section 3.3) correspond to the execution of all the default benchmark configurations defined by PBBS, using either one of our (4) schedulers, or Parlay’s default WS scheduler. All the values presented are calculated from the average of 10 runs. The experiments were carried out in 3 different computing nodes with disparate hardware specifications, presented in Table 1, running Debian GNU/Linux 11 (bullseye), kernel version 5.10.0 and version 10.2.1 of the GNU C compiler.

### Table 1: Computers used in the experimental evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th>Cores/Threads</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel12</td>
<td>2 x Intel Xeon E5-2620 v2</td>
<td>12/24</td>
<td>64 GB DDR3 1600 MHz</td>
</tr>
<tr>
<td>AMD32</td>
<td>4 x AMD Opteron 6272</td>
<td>32/64</td>
<td>64 GB DDR3 1600 MHz</td>
</tr>
<tr>
<td>Intel16</td>
<td>2 x Intel Xeon E5-2609 v4</td>
<td>16/16</td>
<td>32 GB DDR4 2400 MHz</td>
</tr>
</tbody>
</table>

5.1 User-Space LCWS versus Work Stealing

Figure 4 shows the speedups obtained by the User-Space LCWS (USLCWS) implementation against Parlay’s WS scheduler. The box plots depict the distribution of the results obtained for the execution of all benchmark configurations.

The overall conclusion is that USLCWS, for the reasons explained in Section 4, performs worse than WS when the number of processors is close to the number of cores. Although it is able to obtain speedups for some benchmarks, such as (invertedIndex, wikipedia250M, 32) (11% in machine AMD32), and (breadthFirstSearch, rMatGraph_J_12_16000000, n) (10% for \( n = 32 \) in machine AMD32 and 16% for \( n = 16 \) in machine Intel16), (nearestNeighbors, 2DinCube_10000000, 32) (8% in machine AMD32), and (convexHull, 2DinSphere_1000000000, 16) (29% in machine Intel16), among others, the average results for the machine’s number of cores range from ≈ 92% to ≈ 95% of WS’s performance. As it is perceivable in Figure 4, there are several benchmark configurations that, specially on machine AMD32, perform below 75% of WS, reaching even a low of 49% for (histogram, randomSeq_100M_100K_int, 32). These bad results happen essentially on benchmarks that have very small execution times (benchmarks that execute in approximately 20 seconds without using the total amount of threads). USLCWS has a rather slow start, due to the notification mechanism delaying the sharing of work, being, thus, more tailored for more compute-intensive computations.

With fewer resources available, the results are considerably better, with USLCWS consistently gaining against WS. Figure 5 shows that USLCWS obtains speedups higher than 1, across all machines, when the number of processors is less than 50% the number of cores. These speedups range from 2% to 4% on AMD32 and Intel12, and from 0.1% to 6% in Intel16. Moreover, Figure 6 shows that from the 50% mark under, USLCWS obtains speedups for 50% to 80% of the benchmark configurations on AMD32, 68% to 80% on Intel12 and 59% to 78% on Intel16.
In conclusion, the overall gains of USLCWS over WS are in average $\approx 3.8\%$ for AMD32, $\approx 1\%$ for Intel12 and $\approx 1.3\%$ for Intel16. So, the results are promising and already constitute an important step towards our goals: have a scheduler that is synchronization-light and, with that, remove the scheduling overhead when the amount of resources available, and hence parallelism, is low.

If we focus on the best-performing configuration for each benchmark, we are able to achieve speedups ranging from 3.5\% to 25.3\%. Conversely, if we select the least favorable configurations, we observe speed reductions ranging from -0.8\% to -102\%. These sub-optimal configurations often involve a large number of workers and short execution times, which pose challenges for the Low-Cost Work Stealing (LCWS) algorithm, as it takes longer for balancing the load.

5.2 Signal-Based LCWS versus Work Stealing

With the aim of closing the gap to WS when the number of processors is close to the number of cores, we developed the signal-based version of LCWS. The speedup results against Parlay’s WS are presented in Figure 7. Compared with the previous results of USLCWS in Figure 4, we have that the performance for the number of processors equals the number of cores is much better, being in average on a par with WS: 99\% of WS’s performance on AMD32 to $\approx 102\%$ on the remainder machines. For the remainder configurations there is also an overall improvement, making the results clearly superior to the ones of WS, as detailed in the following statistics:

- Speedup greater than 1 for 65\% of the benchmark executions, and gains of 5\%, 10\%, 15\% and 20\% for, respectively, 32\%, 17\%, 7\% and 2\% of the executions on AMD32.
- Speedup greater than 1 for 69\% of the benchmark executions, and gains of 5\%, 10\% and 15\% for, respectively, 27\%, 13\% and 2\% of the executions on Intel12.
- Speedup greater than 1 for 69\% of the benchmark executions, and gains of 5\%, 10\%, 15\% and 20\% for, respectively, 32\%, 18\%, 10\% and 4\% of the executions on Intel16.

These performance gains are evident in the box plots of Figure 7, with most of the boxes being above the 1 threshold and usually having much more elements on the first quartile than on the third. They are also visible in the plots of Figure 5, with an exception for 4 processors in AMD32. The performance on the Intel processor is particularly good, with gains on 60\% and 70\% with the number of processors equals the number of cores, to values that reach the 80\% mark when using fewer processors (such as 2 or 4), observable in Figure 6.

By analyzing the best and worst-performing configurations for each benchmark, similarly to what we did for USLCWS, we observed speedups of 2\% to 22.8\% and speed-downs of 6.80\% to -61\%, respectively. Despite the low occurrence of significant fluctuations in this version’s results, its performance was subpar in specific benchmark configurations with a disproportionately high number of steals, that led to an escalation in the signaling overhead. These configurations
4.3 Signal-Based versus User-Space LCWS

Figures 8a to 8h present a profile of the signal-based LCWS implementation, comparing it to WS and USLCWS. Specifically, Figures 8a and 8b visually confirm the previously observed decrease in memory fences and CAS operations with regard to WS, which was also noted in the preliminary profiling of USLCWS (Figure 3). Further analysis of the results shown in Figures 8e and 8f reveals that the number of memory fences and CAS operations is even lower than what was observed for USLCWS, especially when using 2 workers. This behavior is correlated with the ability of the algorithm to avoid unnecessary work exposures, as shown in Figure 8h, while still maintaining the number of successful steal attempts on par with USLCWS (Figure 8g).

As explained in Section 4, the number of unnecessary work exposures is directly related to the number of memory fences. Therefore, in order to keep local operations synchronization-free, it is crucial to only share work that will be effectively stolen.

5.4 Conservative Exposure and Expose Half

The Conservative Exposure variant (presented in Section 4.1.1) shares the same virtues of the standard signal-based version just evaluated. By also making use of signals, the variant is also able to have a reduced number of unnecessary work exposures, while maintaining USLCWS’s ratio of successful steal attempts.

Accordingly, despite the conservative nature of the algorithm, we can observe that it behaves quite well on all machines, even providing better average speedup results in many configurations (Figure 5). The algorithm demonstrates to be the best option for ≈ 33% of the benchmark configurations, versus the ≈ 49% result of the signal-based version. So, despite not being the fastest on average, it is still very competitive, performing particularly well on benchmarks such as integerSort, wordCounts, invertedIndex, maximalMatching and nearestNeighbors.

Concerning the Expose Half variant (presented in Section 4.1.2), the results show that there is a slight increase in the number of steals and a decrease in the number of idle iterations by thieves. This was to be expected since more tasks are being made public.

The single problem with this strategy is that it also increases the number of steals and a decrease in the number of idle iterations by thieves. Since pop_\texttt{bottom} is the most expensive operation in the deque, this led to execution times not differing much from the single-share versions of the algorithms.

Analyzing the charts on Figures 5 and 6, we may conclude that the impact is short of what was initially expected. Nonetheless, the results for 25% of the cores in Intel16 are promising and may justify further research. We observe speedups of 1.20% to 23.20% and speed-downs of 4.30% to -52%. The configurations with the poorest performance align with those of the standard signal-based
implementation, but the overall results are better. This outcome was anticipated due to the reduced number of signals needed to steal multiple tasks.

6 CONCLUSIONS

In this paper, we presented and evaluated multiple implementations of the Low-Cost Work Stealing (LCWS) algorithm. By integrating the proposed schedulers in the Parlay toolkit and performing a comprehensive evaluation with Problem-Based Benchmark Suite (PBBS), we demonstrated that our signal-based proposals are able to be as good as the Work Stealing (WS) algorithm, when the number of processes comes close to the computer’s number of processing cores, and surpass WS, when the number of processes is diminished to a fraction of the computer’s processing cores.

Our findings indicate that these gains are more influenced by the use of fractions of the computer’s core count rather than raw number of cores, as illustrated in Figures 5 and 6 where the values for 4 cores in computer AMD32 are quite different from the ones for the name number of cores in computers Intel12 and Intel16.

The results obtained push the current state of the art by enabling the efficient use of WS-based load balancing even in the presence of resource managers that, dynamic and adaptively, allocate resources (include processing cores) to co-existing runtime systems.

ACKNOWLEDGMENTS

This work is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT/IP.

REFERENCES


