

# A Verified Parallel Scheduler for OCaml 5

CLÉMENT ALLAIN, INRIA, France

GABRIEL SCHERER, INRIA, France and Université Paris Cité, France

We present the implementation and mechanized verification of a realistic parallel scheduler for OCaml 5 using the Iris-based Zoo framework. Similarly to `Domainlib`, it relies on a work-stealing strategy to perform load balancing but also supports other scheduling strategies thanks to its flexible interface. We provide basic benchmarks demonstrating that its performance is on par with other schedulers from the OCaml ecosystem.

As part of this effort, we verify the Chase-Lev work-stealing deque, as implemented in the `Saturn` library. We show that it features a subtle external and future-dependent linearization point. To deal with it, we introduce new abstractions for reasoning about prophecy variables in Iris.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; **Shared memory algorithms**; • **Software and its engineering** → **Software verification**; *Concurrent programming structures*.

Additional Key Words and Phrases: Verification, Separation Logic, Rocq, Iris, OCaml 5, Concurrent Scheduler, Work-Stealing

## ACM Reference Format:

Clément Allain and Gabriel Scherer. 2026. A Verified Parallel Scheduler for OCaml 5. *Proc. ACM Program. Lang.* 10, PLDI, Article 259 (June 2026), 29 pages. <https://doi.org/10.1145/3808337>

## 1 Introduction

The OCaml programming language migrated from a sequential to a multicore runtime in OCaml 5, imported from the Multicore OCaml research project [Sivaramakrishnan, Dolan, White, Jaffer, Kelly, Sahoo, Parimala, Dhiman and Madhavapeddy 2020] and first released as OCaml 5.0 in 2022. The sequential runtime had a “big runtime lock” guaranteeing that at most one OCaml thread would run at any point in time. The multicore runtime supports “domains” (heavyweight threads) that can run OCaml code in parallel, operating on a shared heap and cooperating for garbage collection. It is designed to offer a M:N threading model with M lightweight tasks (or threads, fibers) are mapped to N domains, with N no larger than the number of CPU cores<sup>1</sup>. Lightweight tasks and their scheduler are left to be implemented in userland, as OCaml libraries running on top of runtime domains.

The authors of the Multicore OCaml runtime implemented the `Domainlib` library for CPU-bound tasks, initially to write benchmarks to test the scalability of the OCaml runtime. It uses a work-stealing scheduler and is state-of-the-art in the OCaml library ecosystem. Other lightweight task libraries include `Moonpool`, which was implemented independently, and `Eio` which focuses on

<sup>1</sup>The multicore runtime of OCaml performs frequent stop-the-world pause in its garbage collector, to facilitate the migration of existing programs using the OCaml foreign function interface — this design does not require adding a read barrier. A consequence of these stop-the-world events is that runtime performance declines sharply if a domain is paused by the operating system, so it is critical to limit the number of domains to available cores: domains are heavier, less composable abstraction than “kernel threads” in typical M:N models. In consequence, they must be controlled by end-user applications, possibly via a concurrency framework that hides them entirely, and software libraries should not implicitly spawn new domains, they need to use a more lightweight task abstraction.

Authors’ Contact Information: Clément Allain, INRIA, Paris, France, [clement.allain@inria.fr](mailto:clement.allain@inria.fr); Gabriel Scherer, INRIA, Paris, France and Université Paris Cité, Paris, France, [gabriel.scherer@inria.fr](mailto:gabriel.scherer@inria.fr).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART259

<https://doi.org/10.1145/3808337>

efficient asynchronous I/O. Those schedulers have been used in performance-sensitive scenarios, benchmarked and optimized, notably using lock-free data structures implemented in OCaml 5.

In this work we present *Parabs*, a verified implementation of a state-of-the-art scheduler for lightweight tasks, following the overall design of *Domainslib*, with some implementation choices inspired by the *Taskflow* C++ library [Huang, Lin, Lin and Lin 2022]. This verification builds on top of the *Zoo* framework [Allain and Scherer 2026], which supports the formal verification of a subset of OCaml 5 in the *Iris* program logic [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018], mechanized within the *Rocq* proof assistant. Our verification effort includes a new mechanized verification of the Chase-Lev work-stealing queue [Chase and Lev 2005], with stronger invariants than had previously appeared in the literature. Our scheduler supports two different scheduling strategies, one using standard randomized work-stealing [Blumofe and Leiserson 1999], the other using work-stealing with *private* dequeues [Acar, Charguéraud and Rainey 2013]. On top of the scheduler, we expose an API closely resembling *Domainslib*.



This work is focused on formal verification but we did write and run relatively simple benchmarks to validate experimentally that the performance of our verified implementation, *Parabs*, is comparable to *Domainslib*; in our tests it is equal or faster.

*Contributions.* Our contributions include:

- (1) A new mechanized verification of the Chase-Lev work-stealing queue [Chase and Lev 2005], with finer-grained invariants than had previously appeared in the literature. In particular, our invariant lets us reason about the failure case of the pop operation, which was missing from earlier formalizations and is essential to prove completion of the work-stealing scheduler when all task queues are exhausted.
- (2) A fully verified implementation of a parallel scheduler in OCaml 5, *Parabs*, which provides a verified alternative to the state-of-the-art *Domainslib* library. To the best of our knowledge, this is the first verified implementation of a parallel work-stealing task scheduler (for any language) using realistic implementation techniques. Our experimental evaluation on a set of simple benchmarks shows that *Parabs* has comparable or better performance than *Domainslib*, and better performance than *Moonpool*.
- (3) At the level of *Iris* proof techniques, we developed extensions of prophecy variables. To reason about the linearization points of our concurrent data structures we needed to introduce “wise” and “multiplexed” prophecy variables, which are reusable building blocks and could be useful for the verification of other concurrent data structures, in any *Iris* formalization of any programming language.

*Artifact.* The *Zoo* verification framework supports a fragment of OCaml called *ZooLang*, with formal semantics defined as an *Iris* program logic, and a partial translator from OCaml to *ZooLang* programs deeply embedded within *Rocq*. Our developments are thus available as OCaml libraries that are readily available for usage, and their *Rocq* specifications, invariants and proofs. (We count 2K lines of OCaml code, and 26K lines of *Rocq* proofs, in the middle of a larger repository.) For reasons of space we cannot possibly hope to describe them in full in the paper, so we focus on the most readily reusable parts: specifications, and key invariants and proof techniques. For more details, we view our code and mechanized proofs as an integral part of this submission; they are available at <https://github.com/clef-men/zoo/tree/pldi26-artifact>, publicly available and open-source.<sup>2</sup> To

<sup>2</sup>For ease of development we followed the *Zoo* approach of working in a mono-repository, so we have an experimental version of *Zoo* with our developments added, as well as some cross-cutting improvements to the pre-existing support libraries and tactics.

ease in-depth exploration, the paper contains direct references to the implementation and the proof as picture/icon links, for example  and .

## 2 Iris Arsenal

*Separation logic.* Iris is a concurrent separation logic [O’Hearn 2007; O’Hearn, Reynolds and Yang 2001; Reynolds 2002] fully mechanized in the Rocq proof assistant [Krebbers, Jourdan, Jung, Tassarotti, Kaiser, Timany, Charguéraud and Dreyer 2018]. As such, it features basic connectives like separation conjunction  $*$  and separating implication  $\multimap$ .

*Persistent assertions.* In Iris, assertions are affine: using a resource consumes it, removes it from the proof context. Some assertions, however, are *persistent*. Once a persistent assertion holds, it holds forever; using it does not consume it. This enables *duplication* ( $P \vdash P * P$ ) and *sharing*. In particular, pure (meta-level) assertions embedded into the logic are persistent.

Formally, persistence is defined in terms of the *persistence modality*:

$$\text{persistent } P \triangleq P \vdash \Box P$$

Informally,  $\Box P$  means  $P$  holds without asserting any exclusive ownership; in other words, it only expresses knowledge. Naturally,  $\Box P$  is persistent.

*Ghost state.* One of the most important features of Iris is its *user-defined higher-order ghost state*, a very flexible form of ghost state. Ghost updates, of the form  $\Rightarrow P$ , allow updating the ghost state during the proof; they are purely logical, hence not visible in the program.

*Sequential specification.* Sequential specifications take the form of Hoare triples:

$$\frac{\left\{ \begin{array}{c} P \\ e \\ \Phi \end{array} \right\}}{\left\{ \Phi \right\}} \qquad \frac{\left\{ \begin{array}{c} \text{stack-model } t \text{ vs} \\ \text{stack\_push } t \text{ v} \end{array} \right\}}{\left\{ () . \text{stack-model } t (v :: \text{vs}) \right\}}$$

where  $P$  is an Iris assertion,  $e$  an expression and  $\Phi$  an Iris predicate over values.

Informally, this triple says: if the precondition  $P$  holds, we can safely execute  $e$  and, if the execution terminates, the returned value satisfies the postcondition  $\Phi$ . It is a persistent resource, allowing executing  $e$  many times.

*Weakest precondition.* Hoare triples are defined using the more primitive notion of *weakest precondition*  $\text{wp } e \{ \Phi \}$ . Informally, it says that: once only, we can execute  $e$  and, if the execution terminates, the returned value satisfies the postcondition  $\Phi$ . Contrary to Hoare triples, it can depend on exclusive ownership and therefore is not persistent.

*Atomic specification.* To specify concurrent operations, we use the notion of *logical atomicity* [da Rocha Pinto, Dinsdale-Young and Gardner 2014]. An operation is said to be logically atomic if it appears to take effect atomically at some point during its execution, called the *linearization point* of the operation. Birkedal, Dinsdale-Young, Guéneau, Jaber, Svendsen and Tzevelekos showed that it implies *linearizability* [Herlihy and Wing 1990] in a sequentially consistent memory model.

In Iris, logical atomicity takes the form of *atomic specifications*:

$$\frac{\left\langle \begin{array}{c} P_{\text{priv}} \\ \bar{x}. P_{\text{pub}} \\ e \\ \bar{y}. Q \\ \Phi \end{array} \right\rangle}{\left\{ \Phi \right\}} \qquad \frac{\left\langle \begin{array}{c} \text{stack-inv } t \\ \text{vs. stack-model } t \text{ vs} \\ \text{stack\_push } t \text{ v} \\ \text{stack-model } t (v :: \text{vs}) \\ () . \text{True} \end{array} \right\rangle}{\left\{ () . \text{True} \right\}}$$

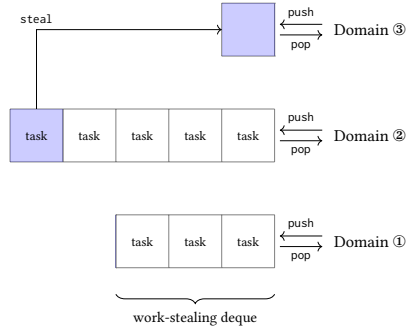


Fig. 1. Work stealing

$P_{priv}$  and  $\Phi$  are standard *private* pre- and postcondition for the user of the specification, similarly to Hoare triples.  $P_{pub}$  and  $Q$  are *public* pre- and postcondition; they specify the linearization point of the operation. Quantifiers  $\bar{x}$  represent the *demonic nature* of  $P_{pub}$ : the exact state at the linearization point, given by  $P_{pub}$ , is unknown until it happens. Quantifiers  $\bar{y}$  represent the *angelic nature* of  $Q$ : at the linearization point, the operation can choose how to instantiate the new state  $Q$ .

In sum, the atomic specification says: if the private precondition  $P_{priv}$  holds, we can safely execute  $e$  and, if the execution terminates, (1) the returned value satisfies the private postcondition  $\Phi$  and (2) at some point during the execution, the state was atomically updated from  $P_{pub}$  to  $Q$ .

*Later modalities.* For technical reasons, Iris specifications involve the *later modality* ( $\triangleright P$ ) [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018], the *big later modality* ( $\blacktriangleright P$ ) and *later credits* ( $\pounds n$ ) [Spies, Gähler, Tassarotti, Jung, Krebbers, Birkedal and Dreyer 2022]. They can be safely ignored at first reading. The later and the big later modalities can be eliminated in one step of computation.

### 3 Chase-Lev Work-Stealing Deque

*Work-stealing.* Randomized *work stealing* [Blumofe and Leiserson 1999] is the standard strategy for parallel task scheduling. It has been implemented in many libraries, including Cilk [Blumofe, Joerg, Kuszmaul, Leiserson, Randall and Zhou 1996; Frigo, Leiserson and Randall 1998], TBB, OpenMP, Taskflow [Huang, Lin, Lin and Lin 2022], Tokio and Domainslib [Multicore OCaml development team 2025].

*Work-stealing deque.* The idea of work-stealing is illustrated in Figure 1. Each domain owns a deque-like data structure, called *work-stealing deque*, to store its tasks. Locally, each domain treats its deque as a stack, operating at the back end. When a domain runs out of tasks, it becomes a thief: it tries to steal a task from the deque of another randomly selected “victim” domain, operating at the front end. Multiple thieves may concurrently attempt to steal tasks from a single deque.

The fact that `steal` works on the opposite end from `push`/`pop` has two purposes. First, the owner domain and a thief more rarely race on the same task. Second, the owner domain pops tasks and pushes smaller sub-tasks, so tasks are typically smaller at the back: the owner pops small tasks that it pushed recently, good for locality, while thieves steal larger tasks, making steals rarer.

*Implementations.* The most popular work-stealing deque algorithm is the Chase-Lev deque [Chase and Lev 2005; Lê, Pop, Cohen and Nardelli 2013]; it is lock-free and unbounded. We verified the implementation from the Saturn library [Karvonen and Morel 2025] 🐘 🍷 along with two other variants: a bounded variant 🐘 🍷, used in the Moonpool [Cruanes 2025] and Taskflow [Huang, Lin, Lin and Lin 2022] libraries, and an idealized infinite-array-based variant 🐘 🍷.

Remarkably, the three variants essentially share the same logical states. In particular, although they do not behave exactly the same way, the original and the idealized versions follow a similar concurrent protocol, involving external and future-dependent linearization.

### 3.1 Interface

The three work-stealing deque implementations below share the same programming interface 🦊:

**val create** : **unit** -> 'a t returns a new deque.

“Private” operations may only be called by the domain owning the deque:

**val size** : 'a t -> **int** returns the number of elements in the deque.

**val is\_empty** : 'a t -> **bool** checks if the deque is empty.

**val push** : 'a t -> 'a -> **unit** pushes an element to the “back” the deque.

**val pop** : 'a t -> 'a **option** pops an element from the “back” of the deque.

Finally, steal is “public” and therefore may be called by any domain:

**val steal** : 'a t -> 'a **option** attempts to steal an element from the “front” of the deque.

### 3.2 Infinite Work-Stealing Deque

**3.2.1 Specification.** The specification of the infinite-array-based version is given in [Figure 2](#). It features three predicates: **inv**, **model** and **owner**.

The persistent assertion **inv**  $t$  represents the knowledge that  $t$  is a valid deque. It is returned by **create** ([INF-WS-DEQUE-CREATE-SPEC](#)) and required by all operations.

The exclusive assertion **model**  $t$   $vs$  represents the ownership of the content of the deque  $vs$ . It is returned by **create** and accessed atomically by all operations.

The exclusive assertion **owner**  $t$   $ws$  represents the owner of the deque;  $ws$  is an upper bound on the current content of the deque ([INF-WS-DEQUE-OWNER-MODEL](#)). It is returned by **create** and used by all private operation: **size** ([INF-WS-DEQUE-SIZE-SPEC](#)), **is\_empty** ([INF-WS-DEQUE-IS-EMPTY-SPEC](#)), **push** ([INF-WS-DEQUE-PUSH-SPEC](#)) and **pop** ([INF-WS-DEQUE-POP-SPEC](#)). The only public operation is **steal** ([INF-WS-DEQUE-STEAL-SPEC](#)), which does not require **owner**.

Note that the public postconditions of the private operations are quite verbose. This is due to the fact that **owner** is passed to the operation and therefore cannot be combined with **model** through [INF-WS-DEQUE-OWNER-MODEL](#) to get information about the content of the deque; instead, we provide such information in the public postcondition.

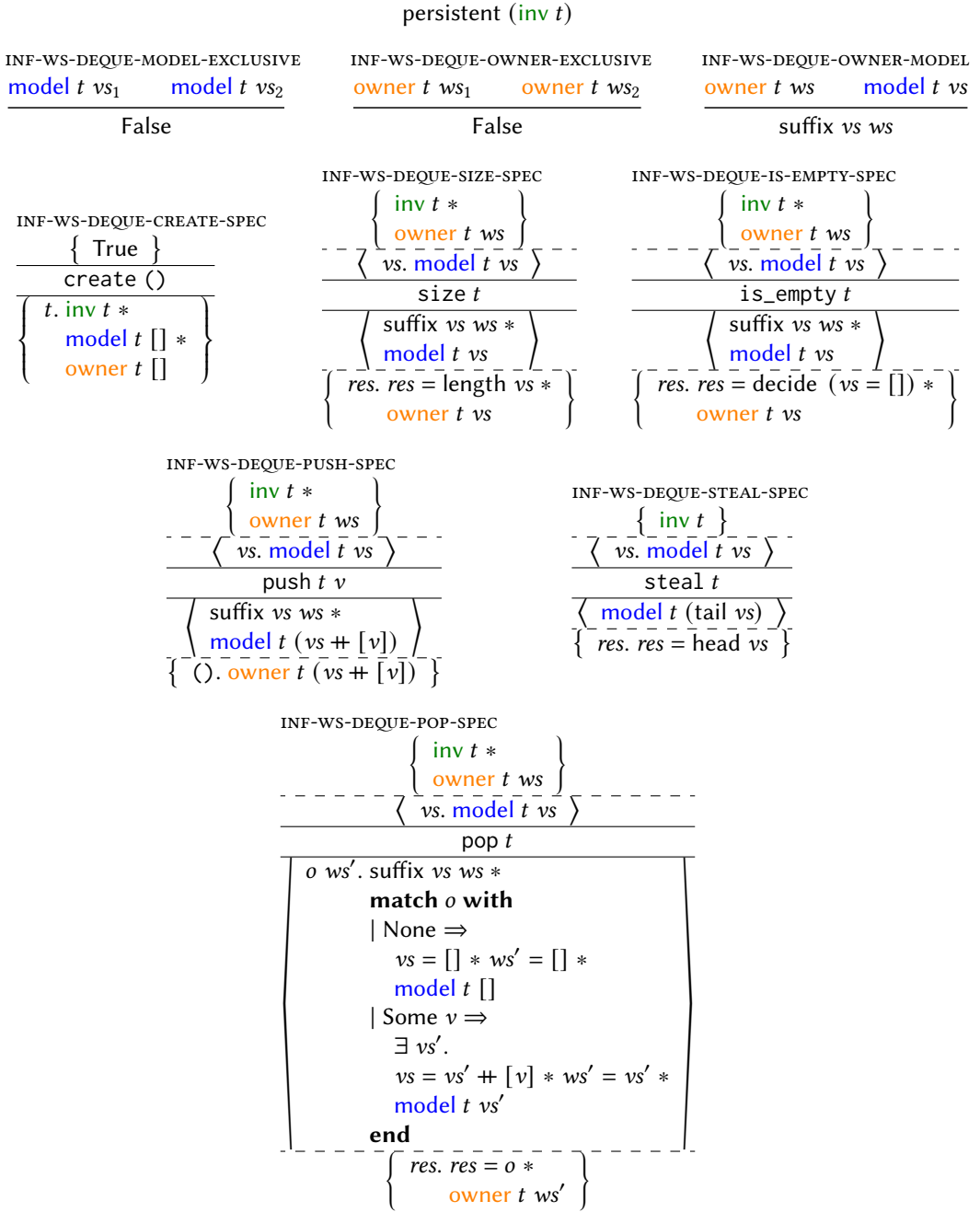
**3.2.2 Weak specification.** [Jung, Lee, Choi, Kim, Park and Kang \[2023\]<sup>3</sup>](#) also worked on the verification of the Chase-Lev work-stealing deque. However, we argue that the specification they prove, given in [Figure 3](#), is unsatisfactory. Indeed, [WS-DEQUE-STEAL-SPEC-WEAK](#) and [WS-DEQUE-POP-SPEC-WEAK](#) say nothing about the observed content of the deque when the operation fails.

In practice, these weaker specifications, especially that of **pop**, are not sufficient to reason about *completion*: once the scheduler terminates, we know that all submitted tasks have finished. In [Section 4](#), we show how our strong specifications are lifted all the way up to the scheduler.

**3.2.3 Implementation.** The implementation relies on (1) an infinite array, (2) a *monotonic* front index for the thieves, and (3) a back index reserved for the owner of the deque.

In general, we can divide the infinite array as in [Figure 4](#). The first part, between 0 and the front index, corresponds to the *persistent* history of stolen values. The second part, between the two indices, corresponds to the logical content of the deque, as represented by **model**. The last part, beyond the back index, corresponds to the private section of the array, reserved for the owner.

<sup>3</sup>See also the master thesis of [Choi \[2023\]](#).

Fig. 2. `Inf_ws_deque`: Specification

Given this representation, the algorithm proceeds as follows. `push t v` writes *v* into the first private cell and atomically increments the back index, thereby publishing the value. Symmetrically, `pop t` atomically decrements the back index and returns the value of the cell it just privatized.

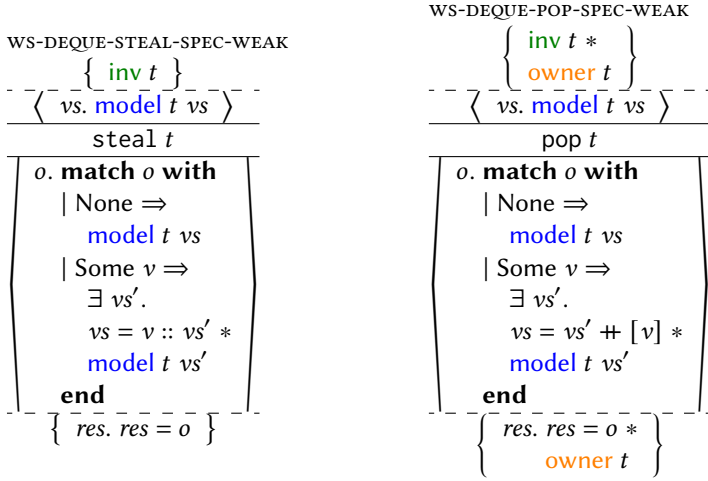


Fig. 3. **Ws\_deque**: Weak specification (excerpt)

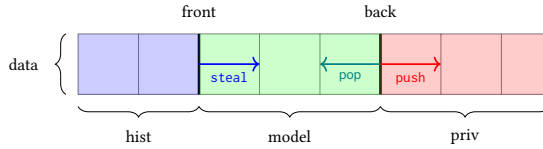


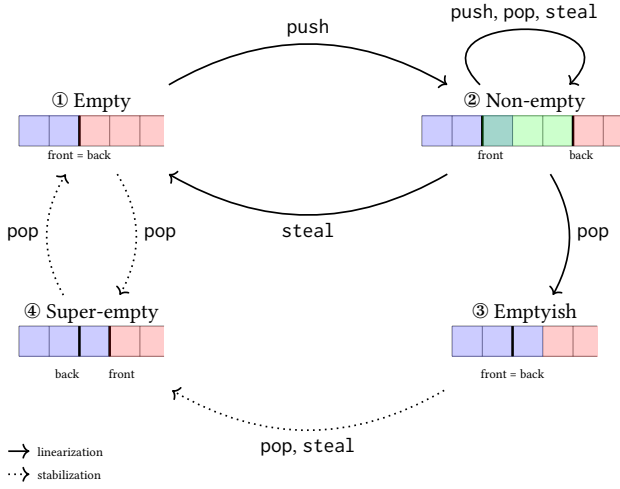
Fig. 4. **Inf\_ws\_deque**: Physical state

steal  $t$  is much more careful: (1) it reads the front and the back indices; (2) if the deque looks empty, it fails; (3) otherwise, it attempts to advance the front index; (4) if the update succeeds, the value at the front index is returned; (5) otherwise, it starts over.

The above description overlooked one crucial aspect: what happens at the limit, when pop and steal compete for the last value in the deque? In that case, the deque must be *stabilized*: pop also attempts to advance the front index before incrementing the back index — whether it wins the update or not — thereby equalizing the two indices.

**3.2.4 Logical states.** Figure 5 tells the same story as above in terms of four *logical states*: (1) in the stable “empty” state, the deque is indeed empty, as indicated by the two equal indices; (2) in the stable “non-empty” state, the **model** is non-empty, meaning thieves may compete for the first value; (3) in the unstable “emptyish” state, the thieves and the owner compete for the same value; (4) in the unstable “super-empty” state, some operation won the value and the deque is waiting to be stabilized by the owner.

Let us now focus on the “emptyish” state. In this physical configuration, it makes sense to say that the **model** of the deque should be empty. In fact, it has to be empty: if a steal operation observed this state, it would conclude that the deque is empty — except under a weak specification. But then, if the **model** should be empty, which operation was linearized during the transition to the “emptyish” state? We have no choice: it should be the winner of the front update, *i.e.* the operation which triggers the transition to the “super-empty” state. In conclusion, we have to predict the winner at each index using a multiplexed prophecy variable (see Section 13).

Fig. 5. `Inf_ws_deque`: Logical state

### 3.3 Bounded Work-Stealing Deque

In the bounded variant, the infinite array is replaced with a finite circular array. As a consequence, tedious reasoning about circular array slices is required. However, the logical states and transitions as well as the prophecy mechanism are essentially the same.

It is an open question whether we could factorize part of the verification through a well-chosen abstraction that could be instantiated both with infinite and circular arrays. One certainty is that this is not possible without slightly altering the implementation of the infinite variant: in `steal`, the front cell is read after performing the update in the infinite variant, which would be incorrect in the finite variant since the owner is allowed to overwrite the value.

### 3.4 Dynamic Work-Stealing Deque

In the original algorithm, the owner may dynamically resize the circular array. More precisely, it can change the array at will provided that the public part (between the two indices) is preserved. Thus, while only one array is stored in the deque, there can be many different circular arrays alive at the same time, *i.e.* accessible by thieves.

While the invariant of Choi [2023] requires additional ghost state to keep track of the arrays and maintain their compatibility, the precision of our notion of logical state allows to only maintain compatibility between the current array and the array read by the next winner (if any).

## 4 Parabs: A Library of Parallel Abstractions

We present the verified Parabs library 🦊 🚚, offering parallel abstractions atop a task scheduler. While it was originally based on `DomainsLib` [Multicore OCaml development team 2025], it evolved as a more ambitious project aimed at unifying various existing paradigms and scheduling strategies. It was designed with a focus on *flexibility*, letting users choose the scheduling strategy and build their own scheduler. One of the motivations of this design is to provide a framework to easily develop and experiment with parallel infrastructures in OCaml 5.

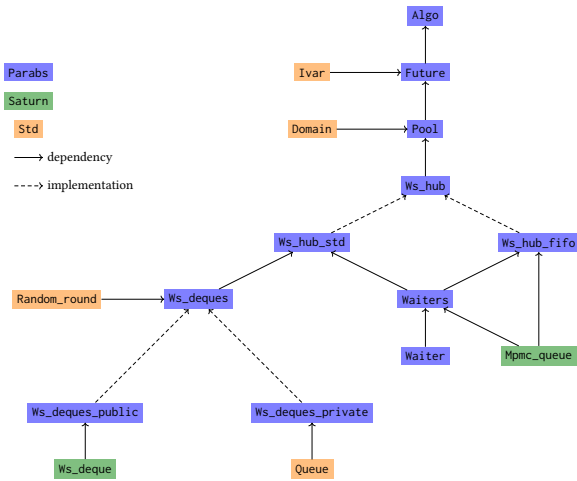


Fig. 6. Overview of the Parabs library

## 5 Overview

Figure 6 gives an overview of Parabs; solid edges represent module dependencies while dashed edges represent interface implementations. Essentially, the library is made of four abstraction levels built on top of each other: **Ws\_deques**, **Ws\_hub**, **Pool** and **Future**.

The **Pool** module provides a task scheduler; internally, it maintains a pool of domains. Its design is inspired by Domainslib, Taskflow [Huang, Lin, Lin and Lin 2022] and Moonpool [Cruanes 2025]. As of today, it supports three scheduling strategies: (1) standard randomized work-stealing [Blumofe and Leiserson 1999] with public dequeues (as presented in Section 3), (2) randomized work-stealing with private dequeues [Acar, Charguéraud and Rainey 2013], (3) a simple “first-in first-out” strategy with one shared queue. In addition, it should be possible to implement other scheduling strategies (see Section 15), e.g. work sharing.

It is worth noting that the three upper levels implemented on top of **Ws\_deques** should be OCaml functors. Unfortunately, ZooLang does not currently support functors; therefore, only one branch of the tree of Figure 6 is active at a time.

## 6 Work-Stealing Deques

At the first level, **Ws\_deques** 🦊 provides a generic interface for a set of work-stealing dequeues, abstracting over the underlying scheduling strategy. It currently has two realizations: **Ws\_deques\_public** (Section 6.1) and **Ws\_deques\_private** (Section 6.2).

### 6.1 Public Deques

The first realization, **Ws\_deques\_public** 🦊 🚚, implements the standard work-stealing strategy with *public dequeues*. More precisely, it simply relies on a shared array of Chase-Lev work-stealing dequeues (see Section 3). These dequeues are public in the sense that both their owner and the thieves can access it directly — which requires synchronization.

## 6.2 Private Deques

The second realization, `Ws_deques_private` 🐼 🛠️, implements the *receiver-initiated* work-stealing algorithm proposed by Acar, Charguéraud and Rainey [2013]<sup>4</sup>. Their idea is to reduce synchronization costs in the fast path of local (owner-only) operations by essentially introducing an indirection. They show that this work-stealing strategy performs well for *fine-grained* parallel programs, *i.e.* when task sizes are small, especially irregular graph computations.

Instead of stealing directly from public deques, thieves follow a protocol: (1) having selected a victim, a thief attempts to send a request by atomically updating the *request cell* of the victim; (2) if the update fails, the thief starts over with another victim, otherwise it awaits a response by repeatedly checking its *response cell*; (3) if the response is negative, the thief starts over, otherwise it returns the task transferred by the victim.

Symmetrically, busy domains regularly poll their request cell and respond accordingly through response cells. Crucially, tasks are stored in private, non-concurrent deques that are only accessed by their owner. In addition, each domain has a *status cell* indicating whether it is (1) blocked, meaning it has no task to share, or (2) non-blocked, meaning it may have tasks to share; before sending a request, thieves check that their victim is non-blocked.

## 7 Waiters

In the realizations of the second level, described in the next section, we use a *sleep-based mechanism* to adapt the number of active thieves. The idea is to put to sleep desperate thieves who do not find work after a number of failed steal attempts. In practice, doing so can improve the overall system performance, especially when tasks are scarce.

To manage sleeping thieves, we use the `Waiters` module 🐼 🛠️. Following the design of Taskflow [Huang, Lin, Lin and Lin 2022], it implements a *two-phase commit protocol*<sup>5</sup> — `Domainslib`<sup>6</sup> relies on a similar mechanism, although it is not as clear-cut.

## 8 Work-Stealing Hub

At the second level, `Ws_hub` 🐼 🛠️ provides a generic interface for a set of tasks supporting work-stealing operations — a so-called “work-stealing hub”. It currently has two realizations: `Ws_hub_std` (Section 8.1) and `Ws_hub_fifo` (Section 8.2).

### 8.1 Work-Stealing Strategy

The first realization, `Ws_hub_std` 🐼 🛠️, implements a standard randomized work-stealing strategy. Under the hood, any work-stealing algorithm may be used, provided that it fits into the `Ws_hub` interface; in particular, it can be instantiated with both realizations of `Ws_deques`.

### 8.2 FIFO Strategy

The second realization, `Ws_hub_fifo` 🐼 🛠️, implements a simple “first-in first-out” scheduling strategy. All workers push and pop tasks from a shared concurrent queue taken from Saturn; thieves also attempt to pop from the queue. Moonpool adopted a similar strategy<sup>7</sup>.

As explained by Cruanes<sup>8</sup>, the point of this strategy is to provide better *latency* than work-stealing — as demanded by certain applications like network servers — at the cost of a lower throughput. Indeed, contrary to work-stealing, older tasks have priority over younger tasks.

<sup>4</sup>They also propose a *sender-initiated* algorithm that we have not implemented.

<sup>5</sup><https://www.1024cores.net/home/lock-free-algorithms/eventcounts>



<sup>6</sup>[https://github.com/ocaml-multicore/domainlib/blob/main/lib/multi\\_channel.ml](https://github.com/ocaml-multicore/domainlib/blob/main/lib/multi_channel.ml)

<sup>7</sup>[https://github.com/c-cube/moonpool/blob/main/src/core/fifo\\_pool.ml](https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.ml)

<sup>8</sup>[https://github.com/c-cube/moonpool/blob/main/src/core/fifo\\_pool.mli](https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.mli)

However, this strategy may also have undesirable consequences. For example, in divide-and-conquer algorithms, this strategy corresponds to *breadth-first* search, whereas work-stealing corresponds to *depth-first* search. On large problems, the former may be unsustainable; on some benchmarks (see Section 12), especially for small cutoffs, Moonpool saturates the memory.

## 9 Pool

At the third level, `Pool`   implements a task scheduler on top of a given realization of `Ws_hub`. It offers essentially the same functionalities as `Domainslib` with a few notable differences. (1) Exceptions raised by tasks are not caught and therefore not re-raised properly by the scheduler since ZooLang does not currently support them. (2) ZooLang does not support algebraic effects [Sivaramakrishnan, Dolan, White, Kelly, Jaffer and Madhavapeddy 2021] either. This limitation imposes a *child-stealing* strategy, as opposed to a *continuation-stealing* strategy that would require capturing the continuation of a computation. Also, this makes it difficult to implement a `yield` operation<sup>9</sup>, *i.e.* an operation that yields control to the scheduler, letting it reschedule the current task later.

### 9.1 Interface

The OCaml interface of the `Pool` module is as follows :

**type** `t` represents a scheduler that owns a pool of worker domains and a set of tasks to compute.

One domain owns the scheduler and can send toplevel tasks: if there are  $n$  workers, then in total  $n + 1$  domains participate to the computation.

**type** `context` represents the runtime execution state of a running scheduler. All asynchronous functions take a `context` parameter, which is passed to asynchronous operations.

**type** `'a task = context -> 'a`

The following functions are used to create a scheduler and run a task on it.

**val** `create` : `int` -> `t` creates a scheduler with the given number of worker domains.

**val** `run_on` : `t` -> `'a task` -> `'a` runs a task on a scheduler. The call returns when the task is finished. Some worker domains may still be executing some tasks asynchronously. (As we see below, `async` can start another task without waiting for its result to terminate.)

**val** `close` : `t` -> `unit` waits for all scheduled tasks to be finished, terminates the worker domains and closes the scheduler.

Finally, the following low-level functions are typically not used directly, as users would prefer the higher-level interface of the `Future` (Section 10) or `Algo` (Section 11) modules:

**val** `size` : `context` -> `int` returns the number of worker domains, which can help when deciding how to split work in an appropriate number of asynchronous tasks.

**val** `async` : `context` -> `unit task` -> `unit` schedules a given task to be executed asynchronously by a running scheduler.

**val** `wait_ivar` : `context` -> (`context`, `'a`) `Ivar.t` -> `unit` waits until the provided `ivar` is set (see below), executing other tasks in the meantime.

*Ivars.* A (`'ctx`, `'a`) `Ivar.t` as used in `Pool.wait_ivar` is a write-once concurrent reference, that contains a value of type `'a` when it is set. It accumulates a list of asynchronous callbacks while it is unset, and invokes them when the variable becomes set:

**val** `create` : `unit` -> (`'ctx`, `'a`) `Ivar.t` creates a fresh `ivar` with no value.

**val** `wait` : (`'ctx`, `'a`) `Ivar.t` -> (`'ctx` -> `'a` -> `unit`) -> `'a option` returns the value if it is set, and accumulates the provided callback otherwise. Despite the name, this function always returns immediately.

<sup>9</sup>`Domainslib` does not currently provide a `yield` operation but it can be easily implemented.

```

val set : 'ctx -> ('ctx, 'a) Ivar.t -> 'a -> unit must be called once per ivar, it
    sets the value and invokes all accumulated callbacks.
val is_set : ('ctx, 'a) Ivar.t -> bool
val get : ('ctx, 'a) Ivar.t -> 'a gets the value of the ivar, assuming it is set.

```

## 9.2 Example

Suppose we have a `fibo : Pool.context -> int -> int` function that computes fibonacci numbers naively but in parallel. Then the following will compute and print the 40-th fibonacci number.

```

let () =
  let pool = Pool.create (Domain.recommended_domain_count () - 1) in
  let fib40 = Pool.run_on pool (fun ctx -> fibo ctx 40) in
  Pool.close pool;
  Printf.printf "fib 40 = %d\n%!" fib40

```

The `fibo` task itself could be defined from the low-level interface directly as follows (assume `fibo_seq` is a sequential definition of the fibonacci function):

```

let rec fibo ctx n =
  if n <= sequential_cutoff then fibo_seq n
  else
    let a = Ivar.create () in
    Pool.async ctx (fun ctx -> Ivar.set ctx a (fibo ctx (n - 1)));
    let b = fibo ctx (n - 2) in
    Pool.wait_ivar ctx a;
    Ivar.get a + b

```

Note that the `fibo` function takes a parameter `ctx : Pool.context`. This is idiomatic, all user functions using **Parabs** for asynchronous computations should take a context parameter. (The higher-level API **Future** and **Algo** also require contexts.) This is common for library-based schedulers: `Domainslib` takes a pool parameter and `Taskflow` takes a `tf::Runtime&` or `tf::Executor&`.

## 9.3 Specification

The specification, in [Figure 7](#), features five predicates: **inv**, **model**, **context**, **finished** and **obligation**.

The persistent assertion **inv**  $t$   $sz$  represents the knowledge that  $t$  is a valid scheduler;  $sz$  is the number of worker domains. It is returned by `create` (**POOL-CREATE-SPEC**) and required only by `size` (**POOL-SIZE-SPEC**). Its only purpose is to record the immutable characteristics of the scheduler.

The assertion **model**  $t$  represents the ownership of scheduler  $t$ . It is returned by `create` (**POOL-CREATE-SPEC**) and required by external operations (**POOL-RUN-SPEC**, **POOL-CLOSE-SPEC**). For example, `run  $t$  task` submits  $task$  to scheduler  $t$ ; it returns both **model** and the output predicate of  $task$ .

The assertion **context**  $t$   $ctx$   $scope$  represents the ownership of *execution context*  $ctx$  attached to scheduler  $t$ ;  $scope$  is a purely logical parameter connecting input and output **context**, which is necessary in the proof. Any task execution happens under such a context (**POOL-RUN-SPEC**, **POOL-ASYNC-SPEC**, **POOL-WAIT-IVAR-SPEC**). In particular, all internal operations require and return **context**. For example, `async  $ctx$  task` submits  $task$  asynchronously while executing under context  $ctx$ ;  $task$  must be shown to execute safely under any context attached to the same scheduler (**POOL-ASYNC-SPEC**).

The persistent assertion **finished**  $t$  represents the knowledge that scheduler  $t$  has finished, meaning all submitted tasks were executed. It can be obtained by calling `close` (**POOL-CLOSE-SPEC**).

$$\begin{array}{c}
\text{persistent (inv } t \text{ sz)} \\
\frac{\text{POOL-INV-AGREE} \quad \text{inv } t \text{ sz}_1 \quad \text{inv } t \text{ sz}_2}{\text{sz}_1 = \text{sz}_2} \\
\\
\text{POOL-CREATE-SPEC} \\
\frac{\{0 \leq \text{sz}\}}{\text{create sz}} \\
\left\{ \begin{array}{l} t. \text{inv } t \text{ sz} * \\ \text{model } t \end{array} \right\} \\
\\
\text{POOL-SIZE-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{inv } t \text{ sz} * \\ \text{context } t \text{ ctx scope} \end{array} \right\}}{\text{size ctx}} \\
\left\{ \begin{array}{l} \text{res. res} = \text{sz} * \\ \text{context } t \text{ ctx scope} \end{array} \right\} \\
\\
\text{POOL-RUN-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{model } t * \\ \forall \text{ ctx scope.} \\ \text{context } t \text{ ctx scope} * \\ \text{wp task ctx } \{ v. \text{context } t \text{ ctx scope} * \Psi v \} \end{array} \right\}}{\text{run } t \text{ task}} \\
\left\{ v. \text{model } t * \Psi v \right\} \\
\\
\text{POOL-ASYNC-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{context } t \text{ ctx scope} * \\ \forall \text{ ctx scope.} \\ \text{context } t \text{ ctx scope} * \\ \text{wp task ctx } \{ \_ . \text{context } t \text{ ctx scope} * \triangleright P * \triangleright \square Q \} \end{array} \right\}}{\text{async ctx task}} \\
\left\{ \begin{array}{l} (). \text{context } t \text{ ctx scope} * \\ \text{consumer } t P * \\ \text{obligation } t Q \end{array} \right\} \\
\\
\text{POOL-WAIT-IVAR-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{context } t \text{ ctx scope} * \\ \text{ivar.inv ivar } \Psi \Xi \Gamma \end{array} \right\}}{\text{wait\_ivar ctx ivar}} \\
\left\{ \begin{array}{l} (). \text{context } t \text{ ctx scope} * \\ \text{ivar.resolved ivar} \end{array} \right\} \\
\\
\text{persistent (obligation } t P) \\
\frac{\text{POOL-CONSUMER-FINISHED} \quad \text{consumer } t P \quad \text{finished } t}{\Leftrightarrow P} \\
\\
\text{persistent (finished } t) \\
\frac{\text{POOL-OBLIGATION-FINISHED} \quad \text{obligation } t P \quad \text{finished } t}{\triangleright \square P} \\
\\
\text{POOL-CLOSE-SPEC} \\
\frac{\left\{ \text{model } t \right\}}{\text{close } t} \\
\left\{ (). \text{finished } t \right\}
\end{array}$$

Fig. 7. Pool: Specification

The persistent assertion **obligation**  $t P$  represents a proof obligation attached to scheduler  $t$ . It allows retrieving  $P$  once  $t$  has finished executing (**POOL-OBLIGATION-FINISHED**). Obligations are obtained by submitting tasks through **async** (**POOL-ASYNC-SPEC**).

The assertion **consumer**  $t P$  represents the right to consume  $P$  once pool  $t$  has finished executing (**POOL-CONSUMER-FINISHED**). Similarly to **obligation**, it can be obtained through **async** (**POOL-ASYNC-SPEC**). Contrary to **obligation**, it is non-persistent and can only be used once through **POOL-CONSUMER-FINISHED**, which is sometimes less convenient. A simple example of use is the parallel “quicksort” algorithm 🐼 🚗, where multiple nested **consumer** assertions are generated.

## 9.4 Implementation

*Worker domains.* The implementation relies on a pool of worker domains and a work-stealing hub. Each worker runs the following loop: (1) get a task using **Ws\_hub**.pop\_steal; (2) if it fails, the scheduler has been closed and so the worker stops, otherwise execute the task in the context of the current worker; (3) start over.

*Blocking.* Care must be taken to block and unblock work-stealing dequeues properly. When the scheduler is closed, it is crucial that workers block their deque before stopping; otherwise, the scheduler may never terminate because of a running worker waiting forever for a response from a stopped but unblocked worker. Also, the main domain, from which tasks can be submitted externally through `run`, must unblock when it is executing tasks and block when it is not.



*Awaiting.* When `wait_ivar` is called and the predicate we are waiting on does not hold, it executes a task from the queue before retrying recursively. In other words, for a given worker the tasks that are currently waiting are stored on the call stack, and they will be re-checked in LIFO order after the current task terminates.

Note: It would be nicer to store waiting tasks in a scheduler data structure; instead of a 'pull' design where waiting tasks check the future result regularly, one could 'push' tasks which become ready when the future they depend on is resolved. `Domainslib` leverages algebraic effects for this: awaiting a future captures the continuation and stores it into the future; when the future is resolved, it resubmits all the waiting tasks.

If no task is available, the worker is put to sleep as mentioned in [Section 7](#); before that it registers a wakeup callback in the corresponding `ivar`, so that it gets woken up if either a new job is pushed or the `ivar` is resolved.

*Shutdown.* In `Domainslib`, scheduler shutdown consists in submitting special tasks through the main domain; when a worker finds such a task, it quickly stops. However, this simple mechanism has at least two drawbacks: (1) it introduces an indirection for every regular task, which may be expensive; (2) it works well under standard work-stealing but is more difficult to implement under other scheduling strategies, especially work-stealing with private dequeues (see [Section 6.2](#)). Consequently, we use an alternative mechanism implemented at the level of `Ws_hub`: a shared flag, regularly checked in `Ws_hub.steal` and `Ws_hub.pop_steal`, is set when the scheduler is closed.

## 10 Futures

At the fourth level, `Future`   implements futures<sup>10</sup>, a standard abstraction for representing the future result of an asynchronous task.

### 10.1 Interface

The OCaml interface of the `Future` module is as follows :

```

type 'a t represents the future result of an asynchronous task, of type 'a.
val return : 'a -> 'a t returns the result of a trivial task that is already finished.
val async : Pool.context -> 'a Pool.task -> 'a t schedules the given task to be executed asynchronously and returns a future for its result.
val wait : Pool.context -> 'a t -> 'a blocks until the result of the given future is available and returns it; the caller domain helps running tasks in the meantime.
val map : Pool.context -> 'a t -> ('a -> 'b) Pool.task -> 'b t schedules a task to run after the result of a future is available and returns a future for its result.
val iter : Pool.context -> 'a t -> ('a -> unit) Pool.task -> unit
  is a specialized map that schedules a 'a -> unit function to be called on the result of a future, when available. The function returns immediately. This can be more efficient as no future is created, but the user code must come with its own synchronization mechanism to know when the function is done.

```

<sup>10</sup>Futures are called *promises* in `Domainslib`. In fact, the two notions are often used in conjunction to represent the two sides of the same object.

## 10.2 Example

We can now write a higher-level version 🦊👉 of the `fibonacci` function from Section 9.1:

```
let rec fibo ctx n =
  if n <= sequential_cutoff then fibo_seq n
  else
    let a = Future.async ctx (fun ctx -> fibo ctx (n - 1)) in
    let b = fibo ctx (n - 2) in
    Future.wait ctx a + b
```

## 10.3 Specification

The specification is given in Figure 8. It features four predicates: `inv`, `result`, `consumer` and `obligation`.

`async` allows submitting a task asynchronously while executing under a context (`FUTURE-SYNC-SPEC`), returning a *future* representing the result of the task. To actually get the result, one must call `wait` (`FUTURE-WAIT-SPEC`). `iter ctx fut task` attaches callback *task* to *fut* (`FUTURE-ITER-SPEC`) and `map ctx fut1 task` creates a new future to be resolved after *fut<sub>1</sub>* (`FUTURE-MAP-SPEC`).

The persistent assertion `inv pool t Ψ ∃` represents the knowledge that *t* is a valid future attached to pool *pool* such that: (1)  $\Psi$  is the *non-persistent output predicate* satisfied by the produced value; (2)  $\exists$  is the *persistent output predicate* satisfied by the produced value.

The persistent assertion `result t v` represents the knowledge that future *t* has been resolved to value *v*. Using `FUTURE-INV-RESULT`, it can also be combined with `inv` to obtain the persistent output predicate. After the pool has finished, it is guaranteed that all futures have been resolved (`FUTURE-INV-FINISHED`).

The assertion `consumer t X` represents the right to consume *X* once future *t* has been resolved. Indeed, using `FUTURE-INV-RESULT-CONSUMER`, it can be combined with `inv` and `result` to obtain *X*. When *t* is created, this assertion is produced with the full non-persistent predicate (`FUTURE-ASYNC-SPEC`, `FUTURE-MAP-SPEC`); then, it can be divided into several parts (`FUTURE-CONSUMER-DIVIDE`). Note that resolution of the future, indicated by `result`, is separated from the division of the output predicates achieved by `consumer`. This lets us resolve different parts of the future's postcondition with different consumer parties. For example, consider a future that initializes an array to be consumed by two separate futures operating on disjoint segments of the array.

The persistent assertion `obligation pool P` represents a proof obligation emitted by `iter` (`FUTURE-ITER-SPEC`). It allows retrieving *P* once *pool* has finished (`FUTURE-OBLIGATION-FINISHED`). Unlike `async` and `map`, `iter` does not return a handle to wait on its result, so its specification requires a different assertion.

## 10.4 Implementation

Futures are implemented using *ivars* (see Section 9.1), as implemented and verified in the Zoo standard library. `async` creates an *ivar* and calls `Pool.async` to resolve it asynchronously. `wait` calls `Pool.wait_ivar` to wait *actively* until the *ivar* is resolved and returns the resulting value.

## 11 Parallel Iterators

On top of `Future`, we implemented and verified standard parallel iterators 🦊👉 that are particularly useful for benchmarks (see Section 12): `for_`, `for_each`, `fold` and `find`. For reasons of space we do not detail all of them, but for example `Algo.for_` has the following type

```
val for_ :
  Pool.context ->
  beg:int (* begin index (included) *) ->
```

$$\begin{array}{c}
\text{persistent (inv pool } t \Psi \Xi) \\
\text{FUTURE-RESULT-AGREE} \\
\frac{\text{result } t \ v_1 \quad \text{result } t \ v_2}{v_1 = v_2}
\end{array}
\quad
\begin{array}{c}
\text{persistent (obligation pool } P) \\
\text{FUTURE-INV-RESULT} \\
\frac{\text{inv pool } t \Psi \Xi \quad \text{result } t \ v}{\Rightarrow \triangleright \square \Xi \ v}
\end{array}
\quad
\begin{array}{c}
\text{persistent (result } t \ v) \\
\text{FUTURE-INV-FINISHED} \\
\frac{\text{inv pool } t \Psi \Xi \quad \text{pool.finished pool}}{\triangleright \exists \ v. \text{result } t \ v}
\end{array}$$

$$\begin{array}{c}
\text{FUTURE-CONSUMER-DIVIDE} \\
\frac{\text{inv pool } t \Psi \Xi \quad \text{consumer } t \left( \lambda v. \bigstar_{X \in X_s} X \ v \right)}{\Rightarrow \bigstar_{X \in X_s} \text{consumer } t \ X}
\end{array}
\quad
\begin{array}{c}
\text{FUTURE-INV-RESULT-CONSUMER} \\
\frac{\text{inv pool } t \Psi \Xi \quad \text{result } t \ v \quad \text{consumer } t \ X}{\Rightarrow \triangleright^2 X \ v}
\end{array}
\quad
\begin{array}{c}
\text{FUTURE-OBLIGATION-FINISHED} \\
\frac{\text{obligation pool } P \quad \text{pool.finished pool}}{\triangleright \square P}
\end{array}$$

$$\begin{array}{c}
\text{FUTURE-ASYNC-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{pool.context pool ctx scope *} \\ \forall \text{ ctx scope.} \\ \text{pool.context pool ctx scope -*} \\ \text{wp task ctx } \left\{ \begin{array}{l} v. \text{pool.context pool ctx scope *} \\ \triangleright \Psi \ v * \\ \triangleright \square \Xi \ v \end{array} \right\} \end{array} \right\}}{\text{async ctx task}} \\
\left\{ \begin{array}{l} t. \text{pool.context pool ctx scope *} \\ \text{inv pool } t \Psi \Xi * \\ \text{consumer } t \ \Psi \end{array} \right\}
\end{array}
\quad
\begin{array}{c}
\text{FUTURE-WAIT-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{pool.context pool ctx scope *} \\ \text{inv pool } t \Psi \Xi \end{array} \right\}}{\text{wait ctx t}} \\
\left\{ \begin{array}{l} v. \ell 2 * \\ \text{pool.context pool ctx scope *} \\ \text{result } t \ v \end{array} \right\}
\end{array}$$

$$\begin{array}{c}
\text{FUTURE-ITER-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{pool.context pool ctx scope *} \\ \text{inv pool } t \Psi \Xi * \\ \forall \text{ ctx scope } v. \\ \text{pool.context pool ctx scope -*} \\ \text{result } t \ v -* \\ \text{wp task ctx } v \\ \left\{ \begin{array}{l} (). \text{pool.context pool ctx scope *} \\ \triangleright \square P \end{array} \right\} \end{array} \right\}}{\text{iter ctx t task}} \\
\left\{ \begin{array}{l} (). \text{pool.context pool ctx scope *} \\ \text{obligation pool } P \end{array} \right\}
\end{array}
\quad
\begin{array}{c}
\text{FUTURE-MAP-SPEC} \\
\frac{\left\{ \begin{array}{l} \text{pool.context pool ctx scope *} \\ \text{inv pool } t_1 \Psi_1 \Xi_1 * \\ \forall \text{ ctx scope } v_1. \\ \text{pool.context pool ctx scope -*} \\ \text{result } t_1 \ v_1 -* \\ \text{wp task ctx } v_1 \\ \left\{ \begin{array}{l} v_2. \text{pool.context pool ctx scope *} \\ \triangleright \Psi_2 \ v_2 * \\ \triangleright \square \Xi_2 \ v_2 \end{array} \right\} \end{array} \right\}}{\text{map ctx } t_1 \ \text{task}} \\
\left\{ \begin{array}{l} t_2. \text{pool.context pool ctx scope *} \\ \text{inv pool } t_2 \Psi_2 \Xi_2 * \\ \text{consumer } t_2 \ \Psi_2 \end{array} \right\}
\end{array}$$

Fig. 8. Future: Specification

```

end_:int (* end index (excluded) *) ->
chunk:int option (* recommended sequential cutoff size *) ->
(int -> int -> unit) Pool.task -> unit

```

and could be used as follows:

```
let double_array ctx t =
```

$$\begin{array}{c}
\text{PROPHET-MODEL-EXCLUSIVE} \\
\frac{\text{model } pid \text{ } prophs_1 \quad \text{model } pid \text{ } prophs_2}{\text{False}} \\
\\
\text{WP-PROPH} \\
\frac{\{ \text{True} \}}{\text{Proph}} \\
\frac{}{\{ pid. \exists prophs. \text{model } pid \text{ } prophs \}} \\
\\
\text{WP-RESOLVE} \\
\frac{\text{atomic } e \quad \text{to-val } e = \text{None} \quad \text{model } pid \text{ } prophs \quad \text{wp } e \left\{ \begin{array}{l} \text{res. } \forall prophs'. \\ \text{prophs} = (\text{res}, v) :: prophs' \text{ } -* \\ \text{model } pid \text{ } prophs' \text{ } -* \\ \Phi \text{ } \text{res} \end{array} \right\}}{\text{wp Resolve } e \text{ } pid \text{ } v \{ \Phi \}}
\end{array}$$

Fig. 9. Reasoning rules for primitive prophets

```

Algo. for_ ctx ~beg:0 ~end:(Array.length t) ~chunk:None
  (fun _ctx begin_ end_ ->
    for i = begin_ to end_ - 1 do
      t.(i) <- 2 * t.(i)
    done)

```

## 12 Benchmarks

We ran several concurrent benchmarks exercising three scheduler implementations: our own Parabs scheduler, the reference Domainslib library, and its alternative Moonpool. The benchmark results validate our qualitative claim that the performance of Parabs is on par with that of Domainslib; we found that it is as efficient, and even slightly faster on some benchmarks.

Due to space constraints, our benchmark results were moved to [Section A](#).

## 13 Prophecy Variables

In 2020, Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs introduced *prophecy variables* in Iris. Essentially, prophecy variables — or *prophets*, as we will call them in this section — can be used to predict the future of the program execution and reason about it. They are key to handle *future-dependent linearization points* [Dongol and Derrick 2014]: linearization points that may or may not occur at a given location in the code depending on a future observation.

In the program, prophecies take the form of two primitives: Proph and Resolve. To reason about them in the logic, Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs [2020] proposed two abstraction layers, which we recall in [Sections 13.1](#) and [13.2](#). To verify the Chase-Lev work-stealing deque (see [Section 3](#)), we needed to introduce two additional layers, presented in [Sections 13.3](#) and [13.4](#).

### 13.1 Primitive Prophet

The first layer consists of *primitive prophets* 🗨️. These prophets are primitive in the sense that they simply reflect the semantics of Proph and Resolve in the program logic. The corresponding reasoning rules are given in [Figure 9](#).

The assertion `model pid prophs` represents the exclusive ownership of the prophet with identifier `pid`; `prophs` is the list of prophecies that must still be resolved.

$$\begin{array}{c}
\text{TYPED-PROPHET-MODEL-EXCLUSIVE} \\
\frac{\text{model } pid \text{ } prophs_1 \quad \text{model } pid \text{ } prophs_2}{\text{False}} \\
\\
\text{TYPED-PROPHET-PROPH-SPEC} \\
\frac{\{ \text{True} \}}{\text{Proph}} \\
\frac{}{\{ pid. \exists prophs. \text{model } pid \text{ } prophs \}} \\
\\
\text{TYPED-PROPHET-RESOLVE-SPEC} \\
\frac{
\begin{array}{c}
\text{atomic } e \quad \text{to-val } e = \text{None} \\
v = \text{prophet.to-val } proph \quad \text{model } pid \text{ } prophs \quad \text{wp } e \left\{ \begin{array}{l}
w. \forall prophs'. \\
prophs = proph :: prophs' \text{ -*} \\
\text{model } pid \text{ } prophs' \text{ -*} \\
\Phi \ w
\end{array} \right\}
\end{array}
}{\text{wp Resolve } e \text{ } pid \ v \{ \Phi \}}
\end{array}$$

Fig. 10. Reasoning rules for typed prophets

**WP-PROPH** says that **Proph** allocates a new prophet with some unknown prophecies to be resolved. **WP-RESOLVE** says that **Resolve**  $e \text{ } pid \ v$  *atomically* resolves the next prophecy of prophet  $pid$ : we learn that the prophecies before resolution  $prophs$  is non-empty and its head is the pair  $(res, v)$  where  $res$  is the evaluation of  $e$ .

### 13.2 Typed Prophet

The second layer consists of *typed prophets* 🏠. They are very similar to primitive prophets except prophecies are now typed. The corresponding reasoning rules, given in **Figure 10**, are essentially the same as before. The prophet must provide a type  $\tau$  along with two functions of-val and to-val. to-val converts an inhabitant of  $\tau$  to a value; **TYPED-PROPHET-RESOLVE-SPEC** relies on it to enforce that the prophecies are well-typed. of-val attempts to convert a value to  $\tau$ ; it is used internally. of-val and to-val must be compatible: of-val (to-val  $proph$ ) = Some  $proph$ .

### 13.3 Wise Prophet

The third layer consists of *wise prophets* 🏠. These prophets *remember* past prophecies. The corresponding reasoning rules are given in **Figure 11**.

The exclusive assertion **model**  $pid \ \gamma \ \text{past } prophs$  represents the ownership of the prophet with identifier  $pid$ ;  $\gamma$  is the logical name of the prophet;  $\text{past}$  is the list of prophecies resolved so far;  $prophs$  is the list of prophecies that must still be resolved.

The persistent assertion **full**  $\gamma \ prophs$  represents the list of all (resolved or not) prophecies associated to the prophet with name  $\gamma$ , as stated by **WISE-PROPHET-FULL-VALID**.

The persistent assertion **snapshot**  $\gamma \ \text{past } prophs$  represents a snapshot of the state of the prophet with name  $\gamma$  at some point in the past. **WISE-PROPHET-SNAPSHOT-VALID** allows to relate the current state of **model** to the past state of **snapshot**.

The persistent assertion **lb**  $\gamma \ lb$  represents a lower bound on the non-resolved prophecies of the prophet with name  $\gamma$ . In particular, as stated by **WISE-PROPHET-LB-VALID**, the list of currently non-resolved prophecies carried by **model** is always a suffix of  $lb$ .

**WISE-PROPHET-RESOLVE-SPEC** is the same as before, except we also update the list of resolved prophecies after resolution.

$$\begin{array}{c}
\text{persistent (full } \gamma \text{ prophs)} \quad \text{persistent (snapshot } \gamma \text{ past prophs)} \quad \text{persistent (lb } \gamma \text{ lb)} \\
\\
\frac{\text{WISE-PROPHET-MODEL-EXCLUSIVE} \quad \text{model pid } \gamma_1 \text{ past}_1 \text{ prophs}_1 \quad \text{model pid } \gamma_2 \text{ past}_2 \text{ prophs}_2}{\text{False}} \quad \frac{\text{WISE-PROPHET-FULL-GET} \quad \text{model pid } \gamma \text{ past prophs}}{\text{full } \gamma \text{ (past } \# \text{ prophs)}} \\
\\
\frac{\text{WISE-PROPHET-FULL-VALID} \quad \text{model pid } \gamma \text{ past prophs}_1 \quad \text{full } \gamma \text{ prophs}_2}{\text{prophs}_2 = \text{past } \# \text{ prophs}_1} \quad \frac{\text{WISE-PROPHET-FULL-AGREE} \quad \text{full } \gamma \text{ prophs}_1 \quad \text{full } \gamma \text{ prophs}_2}{\text{prophs}_1 = \text{prophs}_2} \\
\\
\frac{\text{WISE-PROPHET-SNAPSHOT-GET} \quad \text{model pid } \gamma \text{ past prophs}}{\text{snapshot } \gamma \text{ past prophs}} \quad \frac{\text{WISE-PROPHET-SNAPSHOT-VALID} \quad \text{model pid } \gamma \text{ past}_1 \text{ prophs}_1 \quad \text{snapshot } \gamma \text{ past}_2 \text{ prophs}_2}{\exists \text{ past}_3. \text{past}_1 = \text{past}_2 \# \text{past}_3 * \text{prophs}_2 = \text{past}_3 \# \text{prophs}_1} \\
\\
\frac{\text{WISE-PROPHET-LB-GET} \quad \text{model pid } \gamma \text{ past prophs}}{\text{lb } \gamma \text{ prophs}} \quad \frac{\text{WISE-PROPHET-LB-VALID} \quad \text{model pid } \gamma \text{ past prophs} \quad \text{lb } \gamma \text{ lb}}{\exists \text{ past}_1 \text{ past}_2. \text{past} = \text{past}_1 \# \text{past}_2 * \text{lb} = \text{past}_2 \# \text{prophs}} \\
\\
\frac{\text{WISE-PROPHET-PROPH-SPEC} \quad \{ \text{True} \}}{\text{Proph}} \\
\frac{\{ \text{pid. } \exists \gamma \text{ prophs. model pid } \gamma \text{ [] prophs} \}}{\{ \text{pid. } \exists \gamma \text{ prophs. model pid } \gamma \text{ [] prophs} \}} \\
\\
\frac{\text{WISE-PROPHET-RESOLVE-SPEC} \quad \text{atomic } e \quad \text{to-val } e = \text{None} \quad v = \text{prophet.to-val proph}}{\text{model pid } \gamma \text{ past prophs} \quad \text{wp } e \left\{ \begin{array}{l} w. \forall \text{ prophs}' . \\ \text{prophs} = \text{proph} :: \text{prophs}' -* \\ \text{model pid } \gamma \text{ (past } \# \text{ [proph]) prophs}' -* \\ \Phi w \end{array} \right\}}{\text{wp Resolve } e \text{ pid } v \{ \Phi \}}
\end{array}$$

Fig. 11. Reasoning rules for wise prophets

### 13.4 Multiplexed Prophet

The fourth layer consists of *multiplexed prophets*  $\Rightarrow$ . Essentially, they allow to combine different prophets, each operating at a fixed index. They handle the case when a single prophet is used to make independent predictions, as in Section 3. The corresponding reasoning rules are in Figure 12.

The predicates and rules are basically the same as before, except that (1) **model** now carries sequences of lists of prophecies – one past and one future per index – and (2) **full**, **snapshot** and **lb** are parameterized with an index.

Importantly, the third argument provided to **Resolve** in **WISE-PROPHETS-RESOLVE-SPEC** must be a pair of an index and a prophecy value. Resolution happens only at the given index, meaning the prophecies at other indices are unchanged.

Note that we could generalize this abstraction to non-integer keys. In other words, we could replace sequences with functions of type  $X \rightarrow \tau$ , where  $\tau$  is the prophecy type, and indices with inhabitants of  $X$ . In practice, however, we never needed such generalization.

$$\begin{array}{c}
\text{persistent (full } \gamma \ i \ \text{prophs)} \quad \text{persistent (snapshot } \gamma \ i \ \text{past prophs)} \quad \text{persistent (lb } \gamma \ i \ \text{lb)} \\
\\
\frac{\text{WISE-PROPHETS-MODEL-EXCLUSIVE} \quad \text{model } pid \ \gamma_1 \ \text{pasts}_1 \ \text{prophss}_1 \quad \text{model } pid \ \gamma_2 \ \text{pasts}_2 \ \text{prophss}_2}{\text{False}} \quad \frac{\text{WISE-PROPHETS-FULL-GET} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss}}{\text{full } \gamma \ i \ (\text{pasts } i \ \# \ \text{prophss } i)} \\
\\
\frac{\text{WISE-PROPHETS-FULL-VALID} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss} \quad \text{full } \gamma \ i \ \text{prophs}}{\text{prophs} = \text{pasts } i \ \# \ \text{prophss } i} \quad \frac{\text{WISE-PROPHETS-FULL-AGREE} \quad \text{full } \gamma \ i \ \text{prophs}_1 \quad \text{full } \gamma \ i \ \text{prophs}_2}{\text{prophs}_1 = \text{prophs}_2} \\
\\
\frac{\text{WISE-PROPHETS-SNAPSHOT-GET} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss}}{\text{snapshot } \gamma \ (\text{pasts } i) \ (\text{prophss } i)} \\
\\
\frac{\text{WISE-PROPHETS-SNAPSHOT-VALID} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss} \quad \text{snapshot } \gamma \ i \ (\text{pasts } i) \ (\text{prophss } i)}{\exists \ \text{past}' . \ \text{pasts } i = \text{past} \ \# \ \text{past}' \ * \ \text{prophs} = \text{past}' \ \# \ \text{prophss } i} \quad \frac{\text{WISE-PROPHETS-LB-GET} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss}}{\text{lb } \gamma \ i \ (\text{prophss } i)} \\
\\
\frac{\text{WISE-PROPHETS-LB-VALID} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss} \quad \text{lb } \gamma \ i \ \text{lb}}{\exists \ \text{past}_1 \ \text{past}_2 . \ \text{pasts } i = \text{past}_1 \ \# \ \text{past}_2 \ * \ \text{lb} = \text{past}_2 \ \# \ \text{prophss } i} \\
\\
\frac{\text{WISE-PROPHETS-PROPH-SPEC} \quad \{ \text{True} \}}{\text{Proph}} \\
\frac{\{ \text{pid} . \exists \ \gamma \ \text{prophss} . \text{model } pid \ \gamma \ (\lambda \_ . []) \ \text{prophss} \}}{} \\
\\
\frac{\text{WISE-PROPHETS-RESOLVE-SPEC} \quad \text{atomic } e \quad \text{to-val } e = \text{None} \quad v = \text{prophet.to-val } \text{proph} \quad \text{model } pid \ \gamma \ \text{pasts} \ \text{prophss}}{\text{wp } e \left\{ \begin{array}{l} w . \forall \ \text{prophs} . \\ \text{prophss } i = \text{proph} :: \text{prophs} \ * \\ \text{model } pid \ \gamma \ (\text{alter } (\cdot \ \# \ [\text{proph}]) \ i \ \text{pasts}) \ (\text{prophss } [i \mapsto \text{prophs}]) \ * \\ \Phi \ w \end{array} \right\}} \\
\text{wp Resolve } e \ \text{pid } (i, v) \ \{ \Phi \}
\end{array}$$

Fig. 12. Reasoning rules for multiplexed prophets

## 14 Related Work

*Chase-Lev work-stealing deque.* Jung, Lee, Choi, Kim, Park and Kang [2023]<sup>11</sup> were the first to achieve foundational verification of the Chase-Lev work-stealing deque, including safe memory reclamation schemes. Before, Lê, Pop, Cohen and Nardelli [2013] presented a pen-and-paper proof of the correctness of an ARM implementation and Mutluergil and Tasiran [2019] verified an idealized implementation based on an infinite array using CIVL [Kragl and Qadeer 2021].

As explained in Section 3, however, Jung, Lee, Choi, Kim, Park and Kang [2023] only verify a weak specification, too weak to prove the completion of our scheduler. We verify a strong specification but, contrary to Lê, Pop, Cohen and Nardelli [2013], we rely on a sequentially consistent memory model; extending our work to relaxed memory is left for future work (see Section 15).

<sup>11</sup>See also the master thesis of Choi [2023].

*Parallel schedulers.* To our knowledge, Parabs is the first formally verified realistic work-stealing scheduler. The previous works we are aware of cover toy implementations, not suitable for real-world usage; in contrast, our implementation is close to state-of-the-art schedulers and offers comparable performance according to our preliminary experiments.

Gu, Shao, Chen, Wu, Kim, Sjöberg and Costanzo [2016] is a verified operating system with a thread scheduler; it is realistic but does not attempt to balance threads between CPUs.

De Vilhena and Pottier [2021] verify a simple cooperative scheduler based on algebraic effects, as a case study for their Iris-based program logic. This scheduler does not support parallelism; it runs fibers inside a single domain. Their notion of future/promise is rudimentary; it only supports persistent output predicates. However, their work, especially the way they formalize the scheduler's effects, will be of particular interest when introducing algebraic effects into ZooLang and Parabs.

Ebner, Martínez, Rastogi, Dardinier, Frisella, Ramanandro and Swamy [2025] verify a parallel scheduler with the same interface as Domainslib, which also serves as a case-study for their program logic. However, their implementation is extremely simplified: a task list protected by a mutex. Their notion of future/joinable is also somewhat rudimentary.

*Efficiency guarantees.* Arora, Blumofe and Plaxton [1998] proves complexity bounds for an idealized scheduling policy, using pen-and-paper proofs. We currently have made no attempt to prove complexity bounds for our implementation. A mechanized proof of a scheduling policy exists in an idealized setting in Acar, Charguéraud, Guatto, Rainey and Sieczkowski [2018], but scaling this to a verified implementation would be extremely challenging. In particular, reasoning about the efficiency of concurrent data structures typically requires fairness assumptions, and representing them in Iris specifications is a topic of active research. A very interesting previous work is Hoffmann, Marmor and Shao [2013], which proposes reasoning rules for lock-freedom in a concurrent separation logic for total correctness.

## 15 Future Work

*Relaxed memory model.* The main limitation of our work is inherited from Zoo: it relies on a sequentially consistent memory model whereas OCaml 5 has a relaxed memory model [Dolan, Sivaramakrishnan and Madhavapeddy 2018]. This simplification endangers the soundness of our specifications. Transitioning to relaxed memory by merging Zoo with Cosmo [Mével and Jourdan 2021; Mével, Jourdan and Pottier 2020] involves introducing memory views, which complicates specifications and invariants.

*Language features.* Parabs suffers from the lack of a number of language features unsupported by Zoo. With functors, we could make the Parabs library completely modular. With exceptions, we could catch and re-raise exceptions in **Pool**. With algebraic effects, we could implement *wait* without using the call stack, and implement *continuation stealing* for *async*.

*Other designs.* We could experiment other designs. For instance, one of the two designs of Moonpool relies on a bounded work-stealing deque combined with a master queue. In the literature, many other scheduling strategies were proposed: continuation-stealing [Schmaus, Pfeiffer, Schröder-Preikschat, Hönig and Nolte 2021; Williams and Elliott 2025], steal-half work-stealing [Hendler and Shavit 2002], split work-stealing [Cartier, Dinan and Larkins 2021; Custódio, Paulino and Rito 2023; Dinan, Larkins, Sadayappan, Krishnamoorthy and Nieplocha 2009; Rito and Paulino 2022; van Dijk and van de Pol 2014], idempotent work-stealing [Michael, Vechev and Saraswat 2009].

## Acknowledgments

The implementation of Parabs and the entire verification work was conducted by Clément Allain.

The Chase-Lev work-stealing deque, a key building block, is based on the implementation in the Saturn library [Karvonen and Morel 2025], written by Tom Kelly and further improved by Vesa Karvonen. The scheduler is largely inspired by the Domainslib implementation [Multicore OCaml development team 2025], with also some inspiration from Taskflow [Huang, Lin, Lin and Lin 2022]. Gabriel Scherer implemented and interpreted the benchmarks, documented some interfaces, and improved the scheduler behavior to eliminate a form of busy-waiting on futures.

The verification work of Clément Allain would not have been possible without the work of the Iris development team. Our anonymous reviewers asked pointed questions about the specifications and verified examples, which helped improve the specifications.

The writing the article is joint work from both authors, with large parts derived from the PhD manuscript of Clément Allain.

☯ This is a low-co2 research paper: <https://tcs4f.org/low-co2-v1>. This research was developed, written, submitted and presented without the use of air travel.

## A Benchmarks

In this section, we present simple benchmarks to assess the performance of Parabs relatively to Domainslib [Multicore OCaml development team 2025] and Moonpool [Cruanes 2025] on simple workloads. Benchmarking parallel schedulers is subtle and difficult; we have not tried here to validate and study experimentally all our implementation choices, or to cover the wide range of parallel workloads, but to validate a simple qualitative claim:

For CPU-bound tasks, Parabs has comparable throughput to Domainslib, a state-of-the-art scheduler used in production in the OCaml 5 library ecosystem.

In fact our results validate a stronger qualitative claim: the performance of Parabs are equal or better than Domainslib, with a 10% speedup in some cases.

### A.1 Setting

*A.1.1 Machine.* The benchmark results were produced on a 12-core AMD Ryzen 5 7640U machine, set at a fixed frequency of 2GHz.

*A.1.2 Parameters.* For each benchmark, we pick an input parameter that gives long-enough computation times on our test machine, typically between 200ms and 2s. We use the hyperfine tool and run each benchmark ten time. All benchmark were run with two parameters varying:

- DOMAINS, the number of domains used for computation;
- CUTOFF, representing an input size or chunk size below which a sequential baseline is used.

For each benchmark, we show:

- per-cutoff results with a fixed value DOMAINS = 6, which should be enough to experience scaling issues while not suffering from CPU contention;
- per-domain results with a CUTOFF value that is chosen to work well for all implementations for this benchmark.

Remark: Large cutoff values tend to work well for benchmarks with homogeneous-enough tasks, as they effectively amortize the scheduling costs. The advantage of having schedulers that also perform well on small cutoffs are two-fold. First, this typically indicate that they will adapt to irregular tasks (but: our benchmarks do not perform an in-depth exploration of irregular workloads). Second, this can alleviate the burden of asking users to choose cutoff sizes (by widening the range

of values that perform well), an activity which requires cumbersome hand-tuning and can limit performance portability.

*A.1.3 Scheduler implementations.* Each benchmark is written on top of a simple scheduler interface, for which the following implementations are provided:

- `domainslib` uses the `Domainslib` library;
- `parabs` uses our `Parabs` library;
- `moonpool-fifo` uses the `Moonpool` scheduler with a global FIFO queue of task;
- `moonpool-ws` uses the `Moonpool` scheduler with a work-stealing pool of tasks, which is described as better for throughput
- `sequential` is a baseline implementation with no parallelism, all tasks run sequentially on a single domain.

We used the latest software versions currently available: `Domainslib` 0.5.2, and `Moonpool` 0.9.

*A.1.4 Benchmarks.*

`fibonacci` 🐘. A parallel implementation of Fibonacci extended with a sequential cutoff: below the cutoff value, a sequential implementation is used.

`iota` 🐘. This benchmark uses a parallel-for to write a default value in each cell of an array. We expect significant variations due to the `CUTOFF` parameter.

`for_irregular` 🐘. This benchmark uses a parallel-for loop with irregular per-element workload: as a first approximation, the  $i$ -th iteration computes `fibonacci i`; this cost grows exponentially in  $i$ , so the majority of computation work is concentrated on the largest loop indices.

`lu` 🐘. This benchmark performs the LU factorization of a random matrix of floating-point values. It consists in  $O(N)$  repetitions of a parallel-for loop of  $O(N)$  iterations, where each iteration performs  $O(N)$  sequential work.

`matmul` 🐘. This benchmark computes matrix multiplication with a very simple parallelization strategy — only the outer loop is parallelized. In other word, there is a parallel-for loop with  $O(N)$  iterations, where each iteration performs  $O(N^2)$  sequential work work.

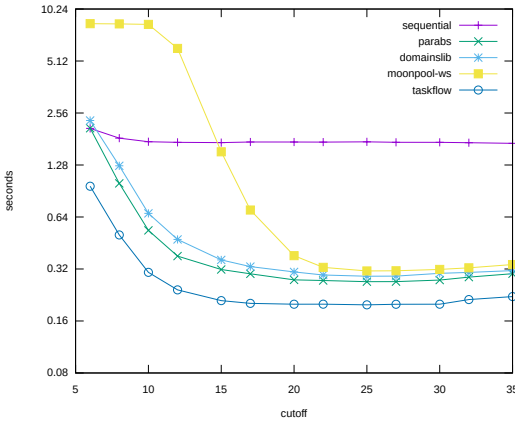
## A.2 Results

*A.2.1 Pre-benchmarking expectations.* Our expectation before running the benchmarks is that `Parabs` has the same performance as `Domainslib`, and that they are both more efficient than `Moonpool` (which uses a central pool of jobs instead of per-domain dequeues).

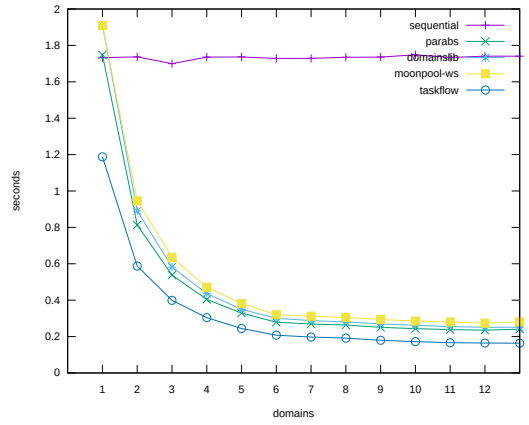
Because `Moonpool` has a less optimized scheduler, we expect scheduling overhead to be an issue for small `CUTOFF` values.

On all schedulers, the performance for larger `CUTOFF` values should be good if the benchmark has homogeneous/regular tasks, and it should be worse if the benchmark has heterogeneous/irregular tasks.

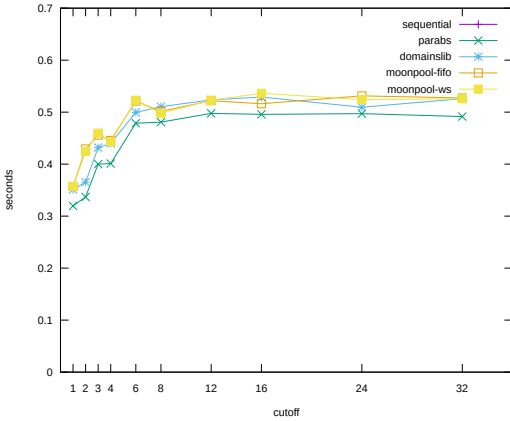
*A.2.2 Per-benchmark results.* [Figure 13](#) and [Figure 13](#) contain the full results, with per-cutoff and per-domain plots for each benchmarks. Notice that while the per-domain plot always use linear axes, the per-cutoff plots often use logarithmic plot axes, to preserve readability when performance difference become very large for small cutoff values, and to express large ranges of possible cutoff choices.



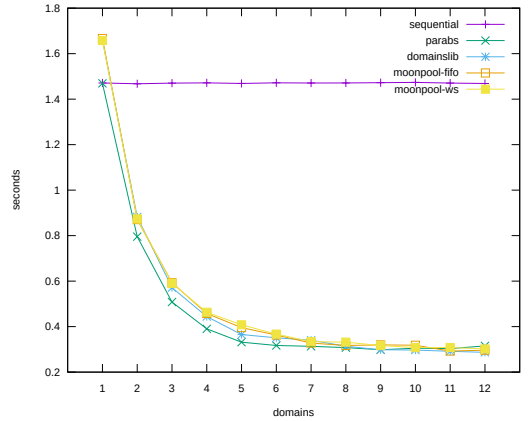
(a) fibonacci: varying CUTOFF



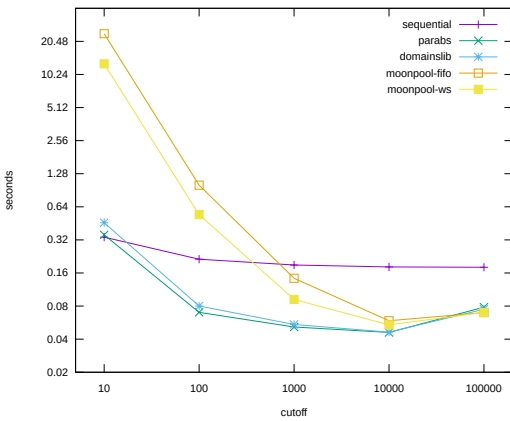
(b) fibonacci: varying DOMAINS



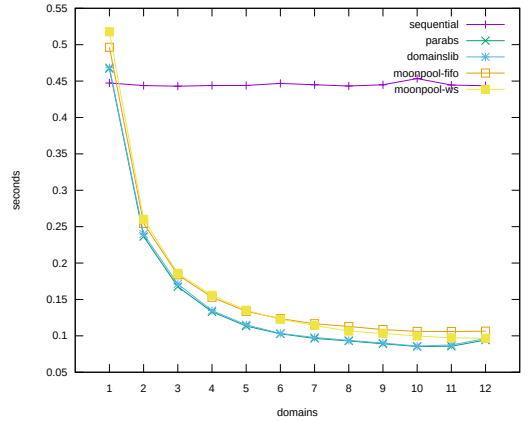
(c) for\_irregular: varying CUTOFF



(d) for\_irregular: varying DOMAINS

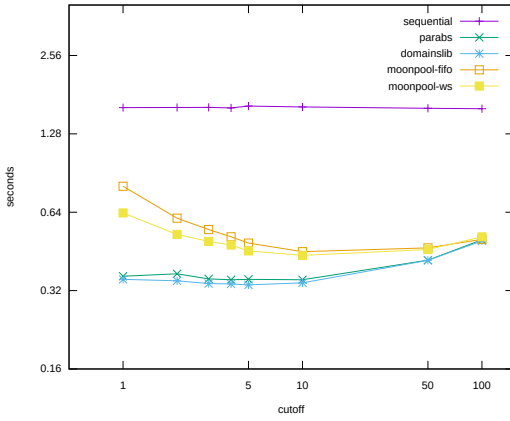


(e) iota: varying CUTOFF

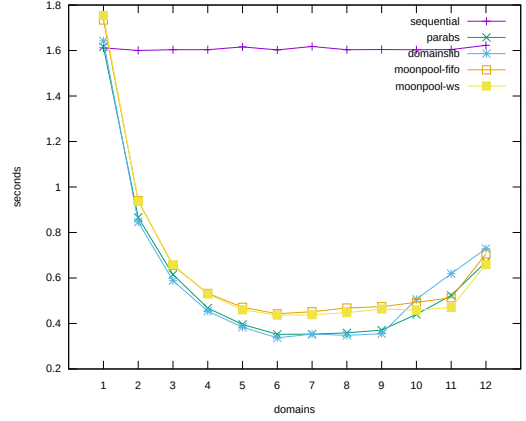


(f) iota: varying DOMAINS

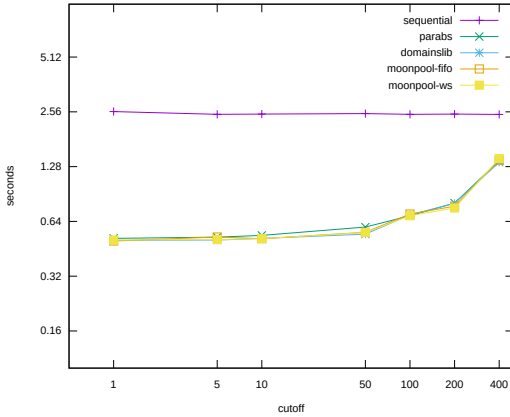
Fig. 13. Benchmarks (1/2)



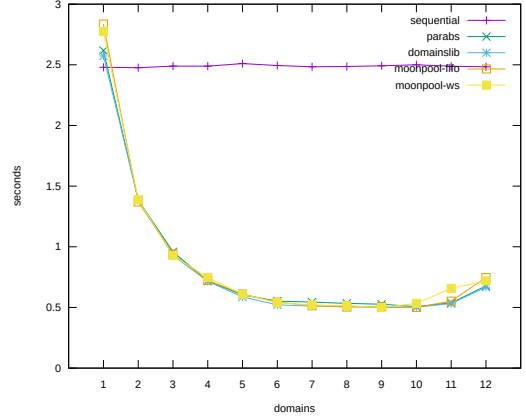
(g) lu: varying CUTOFF



(h) lu: varying DOMAINS



(i) matmul: varying CUTOFF



(j) matmul: varying DOMAINS

Fig. 13. Benchmarks (2/2)

fibonacci. As a control, we introduced a Taskflow implementation for this benchmark, which is a C++ version using the Taskflow scheduler. We expect Taskflow to be faster due to a more optimized, industrial implementation and a more aggressively optimizing compiler (clang++ -O3).

In the per-cutoff results (logarithmic scale), we see that all schedulers start to behave badly when the CUTOFF becomes small enough, with exponentially-decreasing performance after a certain drop point. For Moonpool, performance drops around CUTOFF = 20. The FIFO and work-stealing variants have similar profiles, with work-stealing performing noticeably better. We put a timeout at 8 seconds, so the results on small cutoffs are clamped. For Parabs, Domainslib and Taskflow, performance drops around CUTOFF = 15. Parabs performs noticeably better than Domainslib for small-enough cutoff values, and Taskflow is even better: for CUTOFF = 12, Taskflow is 51% faster than Parabs which is in turn 22% faster than Domainslib. In fact, even for the sequential scheduler we observe a small performance drop: the task-using version creates closures and performs indirect calls, so it is noticeably slower (by a constant factor) than the version used below the sequential cutoff.

Note: we observe very large memory usage with Moonpool at smaller cutoff values — when computing fibonacci 40, attempting to run the benchmark with CUTOFF = 5 fails with out-of-memory errors on a machine with 32Gio of RAM. This seems to come from the FIFO architecture which runs the oldest and thus biggest task first, and thus stores an exponential number of smaller tasks in the queue.

Per-domain results (linear scale): we studied per-domain performance on a CUTOFF = 25 point where all implementations behave well. For this value we see relatively few performance difference between OCaml implementations, especially for larger number of domains ( $\text{DOMAINS} \geq 7$ ). For example for 4 domains, Taskflow is 32% faster than Parabs, which is 6% faster than Domainslib, which is 4% faster than Moonpool.

for\_irregular. This benchmark is designed to behave poorly with large CUTOFF values. We indeed observe better noticeably performance with CUTOFF = 1 than with larger values, across all schedulers — for example domainslib is 50% slower with CUTOFF = 8.

In the per-cutoff results we observe that parabs performs best on this benchmark, then domainslib, then moonpool.

In the per-domain results (with CUTOFF = 1) we see that parabs performs noticeably better than the other implementations for relatively low domain counts, and they become comparable around  $\text{DOMAINS} \geq 7$ .

iota. Each iteration of parallel-for in iota is immediate, so as expected we observe a large sensitivity to the choice of CUTOFF, with parabs and domainslib performing much better than moonpool on smaller CUTOFF values.

In the per-domains result we see that domainslib and parabs have similar performance, noticeably better than the moonpool implementations.

lu. The performance is relatively stable over most choices of CUTOFF. The per-domain results are similar across all benchmarks after controlling for the one-domain shift of Moonpool.

Remark: we observe a marked decline in performance, across all schedulers, when the number of domains becomes close to the number of available cores, around  $\text{DOMAINS} \geq 10$ . We believe that this comes from the high-allocation rate of this benchmark (10.2GiB/s) causing frequent minor collections, and thus stop-the-world pauses, with some domains temporarily suspended by the operating system. In other words, the slowdown comes from the OCaml runtime, not from the scheduler implementations. The allocations can be avoided in this benchmark by optimizing more aggressively to eliminate float boxing, but this phenomenon is likely to occur for other high-allocation OCaml programs so we chose to preserve it.

matmul. The performance is stable across a wide range of CUTOFF values. The parallel-loop performs 500 iterations, so CUTOFF values closer to 500 prevent parallelization and bring performance closer to the sequential scheduler.

The per-domain performance is remarkably similar under all schedulers: our implementation of matrix multiplication has a coarse-grained parallelization strategy where the choice of scheduler makes no difference.

*A.2.3 Result summary.* Overall, Parabs has the same qualitative performance as Domainslib. In fact it performs measurably better (around 10% better for some domain values) on the benchmarks fibonacci and for\_irregular, which have irregular tasks; and it has qualitatively the same performance otherwise.

## References

- Umüt A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat scheduling: provable efficiency for nested parallelism. *SIGPLAN Not.* 53, 4 (June 2018), 769–782. doi:10.1145/3296979.3192391
- Umüt A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private dequeues. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 219–228. doi:10.1145/2442516.2442538
- Clément Allain and Gabriel Scherer. 2026. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *Proc. ACM Program. Lang.* 10, POPL (2026). <https://clef-men.github.io/publications/allain-scherer-26.pdf>
- Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) (SPAA '98). Association for Computing Machinery, New York, NY, USA, 119–129. doi:10.1145/277651.277678
- Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473586
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distributed Comput.* 37, 1 (1996), 55–69. doi:10.1006/JPDC.1996.0107
- Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748. doi:10.1145/324133.324234
- Hannah Cartier, James Dinan, and D. Brian Larkins. 2021. Optimizing Work Stealing Communication with Structured Atomic Operations. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 36:1–36:10. doi:10.1145/3472456.3472522
- David Chase and Yossi Lev. 2005. 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*, Phillip B. Gibbons and Paul G. Spirakis (Eds.). ACM, 21–28. doi:10.1145/1073970.1073974
- Jaemin Choi. 2023. Formal Verification of Chase-Lev Deque in Concurrent Separation Logic. *CoRR* abs/2309.03642 (2023). arXiv:2309.03642 doi:10.48550/ARXIV.2309.03642
- Simon Cruanes. 2025. Moonpool. <https://github.com/c-cube/moonpool>
- Rafael Custódio, Hervé Paulino, and Guilherme Rito. 2023. Efficient Synchronization-Light Work Stealing. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, Kunal Agrawal and Julian Shun (Eds.). ACM, 39–49. doi:10.1145/3558481.3591099
- Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. doi:10.1007/978-3-662-44202-9\_9
- Paulo Emilio de Vilhena and Francois Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434314
- James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM. doi:10.1145/1654059.1654113
- Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 242–255. doi:10.1145/3192366.3192421
- Brijesh Dongol and John Derrick. 2014. Verifying linearizability: A comparative survey. *CoRR* abs/1410.6268 (2014). arXiv:1410.6268 <http://arxiv.org/abs/1410.6268>
- Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1516–1539. doi:10.1145/3729311
- Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. 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*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 212–223. doi:10.1145/277650.277725
- Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (Savannah, GA, USA) (OSDI'16)*. USENIX Association, USA, 653–669.
- Danny Hendler and Nir Shavit. 2002. 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*, Aleta Ricciardi

- (Ed.). ACM, 280–289. doi:10.1145/571825.571876
- Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972
- Jan Hoffmann, Michael Marmor, and Zhong Shao. 2013. Quantitative Reasoning for Proving Lock-Freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*. IEEE Computer Society, USA, 124–133. doi:10.1109/LICS.2013.18
- Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distributed Syst.* 33, 6 (2022), 1303–1320. doi:10.1109/TPDS.2021.3104255
- Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 828–856. doi:10.1145/3622827
- Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151
- Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. doi:10.1145/3371113
- Vesa Karvonen and Carine Morel. 2025. Saturn. <https://github.com/ocaml-multicore/saturn>
- Bernhard Kragl and Shaz Qadeer. 2021. The CiviL Verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19–22, 2021*. IEEE, 143–152. doi:10.34727/2021/ISBN.978-3-85448-046-4\_23
- Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. doi:10.1145/3236772
- Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23–27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 69–80. doi:10.1145/2442516.2442524
- Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473571
- Glen Mével, Jacques-Henri Jourdan, and Francois Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. doi:10.1145/3408978
- Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. 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*, Daniel A. Reed and Vivek Sarkar (Eds.). ACM, 45–54. doi:10.1145/1504176.1504186
- Multicore OCaml development team. 2025. Domainlib. <https://github.com/ocaml-multicore/domainlib>
- Suha Orhun Mutluergil and Serdar Tasiran. 2019. A mechanized refinement proof of the Chase-Lev deque using a proof system. *Computing* 101, 1 (2019), 59–74. doi:10.1007/S00607-018-0635-4
- Peter W. O’Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1–3 (2007), 271–307. doi:10.1016/J.TCS.2006.12.035
- Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10–13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. doi:10.1007/3-540-44802-0\_1
- John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817
- Guilherme Rito and Hervé Paulino. 2022. Scheduling computations with provably low synchronization overheads. *J. Sched.* 25, 1 (2022), 107–124. doi:10.1007/S10951-021-00706-6
- Florian Schmaus, Nicolas Pfeiffer, Wolfgang Schröder-Preikschat, Timo Hönig, and Jörg Nolte. 2021. Nowa: A Wait-Free Continuation-Stealing Concurrency Platform. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17–21, 2021*. IEEE, 360–371. doi:10.1109/IPDPS49936.2021.00044
- K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. doi:10.1145/3408995
- K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design*

*and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. doi:10.1145/3453483.3454039

- Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. doi:10.1145/3547631
- Tom van Dijk and Jaco C. van de Pol. 2014. Lace: Non-blocking Split Deque for Work-Stealing. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 8806)*, Luís M. B. Lopes, Julius Zilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander (Eds.). Springer, 206–217. doi:10.1007/978-3-319-14313-2\_18
- Conor John Williams and James Elliott. 2025. Libfork: Portable Continuation-Stealing With Stackless Coroutines. *IEEE Trans. Parallel Distributed Syst.* 36, 5 (2025), 877–888. doi:10.1109/TPDS.2025.3543442

Received 2025-11-14; accepted 2026-04-03