A verified parallel scheduler for OCaml 5

CLÉMENT ALLAIN, INRIA, France
GABRIEL SCHERER, INRIA, France

We present the implementation and mechanized verification of a realistic parallel scheduler for OCaml 5 using the Iris-based Zoo framework. Similarly to `Domainslib`, 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.

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[1]. The implementation of lightweight tasks and their scheduler is left to be done in userland, as an OCaml library running on top of runtime-provided domains.

The authors of the Multicore OCaml runtime implemented the `Domainslib` 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 efficient asynchronous I/O. All of 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],

[1]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 a 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; Gabriel Scherer, INRIA, Paris, France, gabriel.scherer@inria.fr.

the other using work-stealing with *private* deques [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.[2] To 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 $-\!\!*$.

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.

[2]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.

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

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

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 $\Rrightarrow P$, 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:

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

where P is an Iris assertion, e an expression and Φ 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 Φ. It is a persistent resource, allowing executing e many times.

Weakest precondition. Hoare triples are defined using the more primitive notion of *weakest precondition* $\mathsf{wp}\ e\ \{\ \Phi\ \}$. Informally, it says that: once only, we can execute e and, if the execution terminates, the returned value satisfies the postcondition Φ. 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; this point is called the *linearization point* of the operation. Birkedal, Dinsdale-Young, Guéneau, Jaber, Svendsen and Tzevelekos showed that this notion implies *linearizability* [Herlihy and Wing 1990] in a sequentially consistent memory model.

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

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

P_{priv} and Φ 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 $\overline{x}$ represent the *demonic nature* of P_{pub}: the exact state at the linearization point, given by P_{pub}, is unknown until it happens. Quantifiers $\overline{y}$ 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 Φ 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 sometimes involve the *later modality* ($\triangleright\ P$) [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018], the *big later modality* ($\blacktriangleright\ P$) and *later credits* ($£\ n$) [Spies, Gäher, Tassarotti, Jung, Krebbers, Birkedal and Dreyer 2022]. They can be

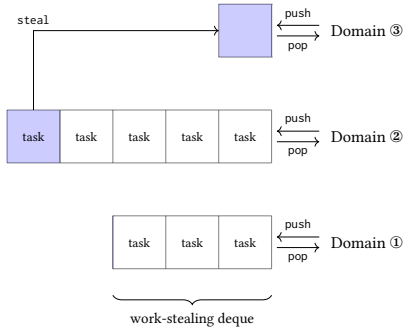


Fig. 1. Work stealing

safely ignored at first reading. Both 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, illustrated in Figure 1, is the following. 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 we describe 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 it 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][3] 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, contrary to our specification, 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 the *completion* of a work-stealing scheduler: 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.

Another point we would like to make is that weakening the specification does make the verification simpler, but one may argue that the most subtle and interesting part of it is lost.

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.

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. `steal` t is much more careful: (1) it reads the front and the back indices; (2) if the deque looks

[3] See also the master thesis of Choi [2023].

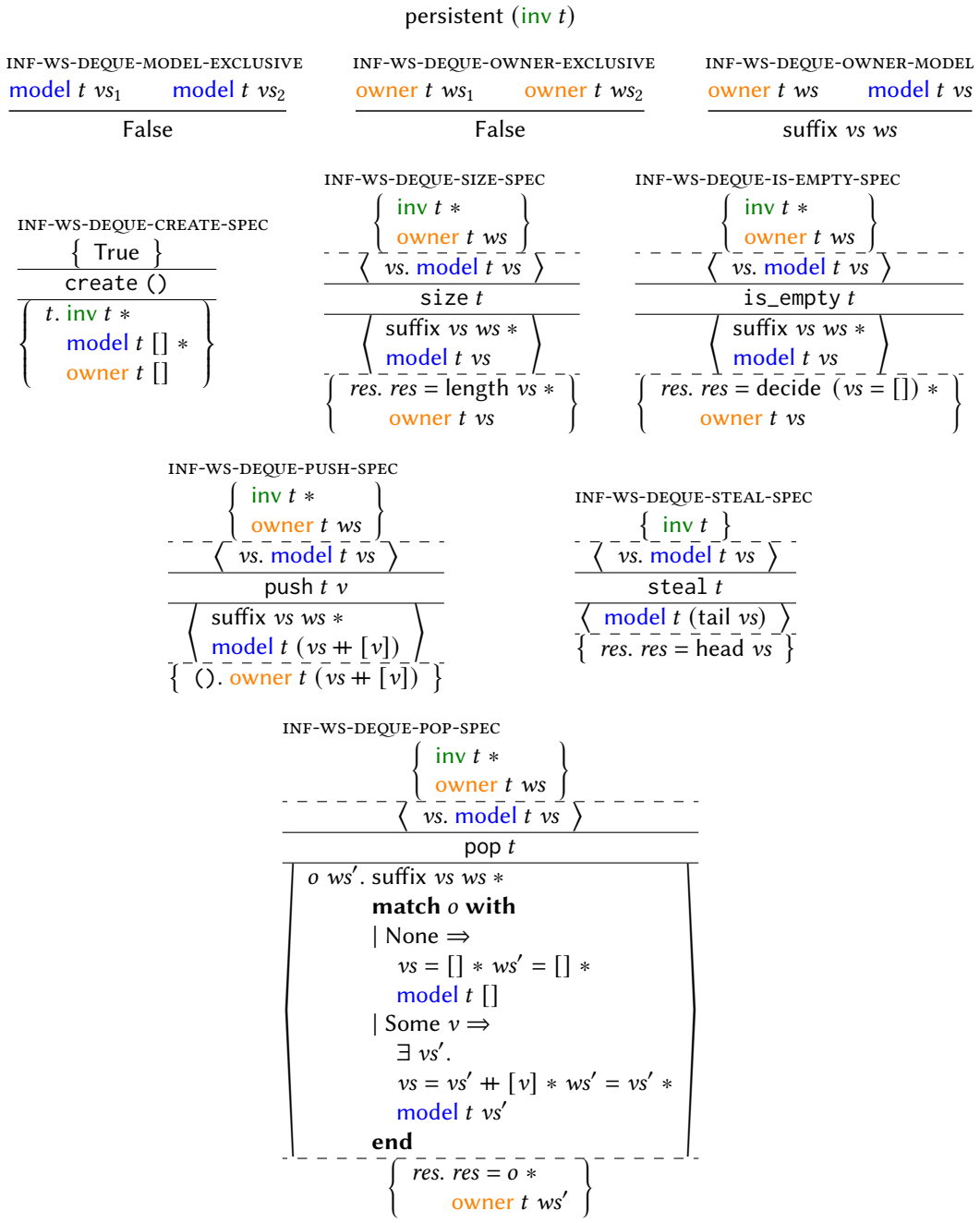


Fig. 2. **`Inf_ws_deque`**: Specification

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.

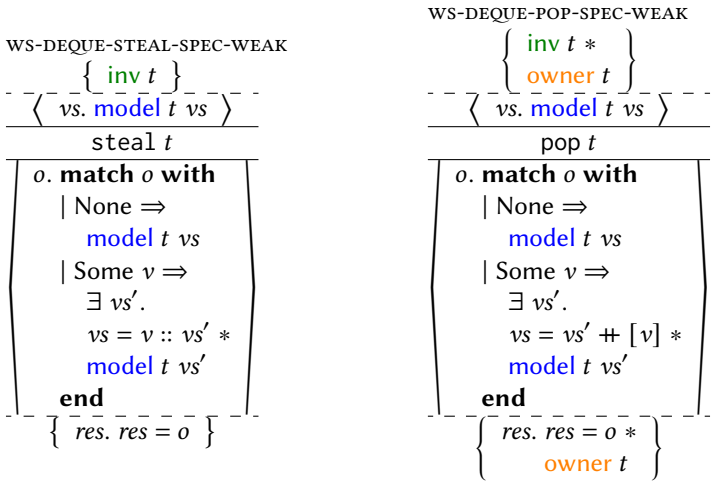


Fig. 3. **`Ws_deque`**: Weak specification (excerpt)

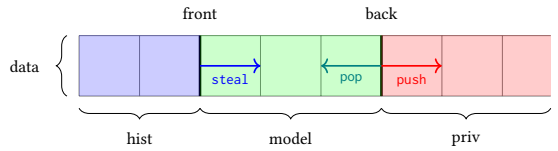


Fig. 4. **`Inf_ws_deque`**: Physical state

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, is 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).

3.3 Bounded work-stealing deque

In the bounded variant, the infinite array is replaced with a finite circular array. As a consequence, the convenient infinite representation goes away and tedious reasoning about circular array slices

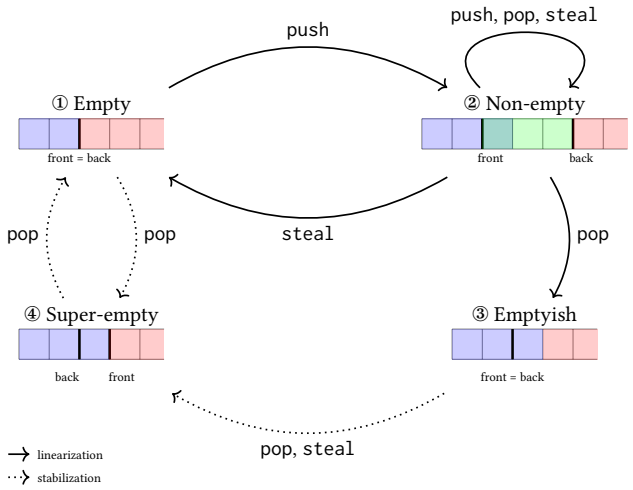


Fig. 5. **`Inf_ws_deque`**: Logical state

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.

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`**.

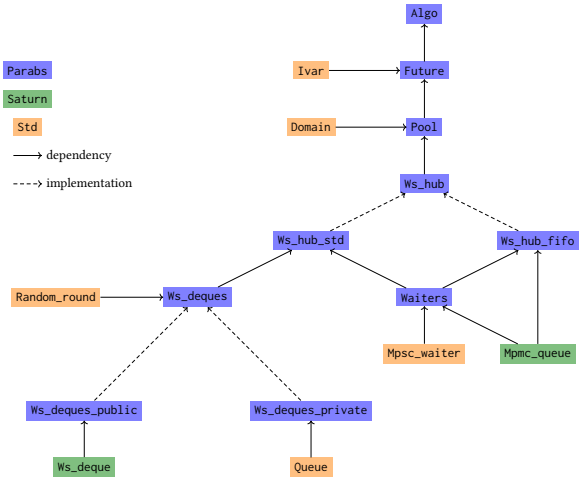


Fig. 6. Overview of the Parabs library

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 deques (as presented in Section 3), (2) randomized work-stealing with private deques [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 deques, 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 deques*. More precisely, it simply relies on a shared array of Chase-Lev work-stealing deques (see Section 3). These deques 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][4]. 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.

[4]They also propose a *sender-initiated* algorithm that we have not implemented.

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*[5] — `Domainslib`[6] 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 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 attempts to pop from the queue. `Moonpool` adopted a similar strategy[7].

As explained by Cruanes[8], 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.

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.

[5] https://www.1024cores.net/home/lock-free-algorithms/eventcounts
[6] https://github.com/ocaml-multicore/domainslib/blob/main/lib/multi_channel.ml
[7] https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.ml
[8] https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.mli

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) Since ZooLang does not support algebraic effects [Sivaramakrishnan, Dolan, White, Kelly, Jaffer and Madhavapeddy 2021] either, the interface is slightly more involved (see *execution contexts* in Section 9.3).

Moreover, 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[9], *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 n worker domains and a set of tasks to compute. One domain owns the scheduler and can send toplevel tasks, so 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 the high-level part of the module, needed to create a scheduler and run a toplevel task.

- `val create : int -> t` creates a scheduler with the given number of worker domains.
- `val run : t -> 'a task -> 'a` runs a task on a scheduler. The call returns when the task is finished.
- `val kill : t -> unit` stops a scheduler, terminating workers when they become idle. This should be called once you know that the toplevel task is finished.

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 to define asynchronous tasks.

- `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_until : context -> (unit -> bool) -> unit` waits until the provided predicate returns `true`, executing other tasks in the meantime.

9.2 Example

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

```
let () =
  let pool = Pool.create (Domain.recommended_domain_count () - 1) in
  let fib40 = Pool.run pool (fun ctx -> fibo ctx 40) in
  Pool.kill pool;
```

[9] `Domainslib` does not currently provide a `yield` operation but it can be easily implemented.

```
  Printf.printf "fib 40 = %d\n%!" fib40
```

The `fibo` 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 = Atomic.make None in
    Pool.async ctx (fun ctx -> Atomic.set a (Some (fibo ctx (n - 1))));
    let b = fibo ctx (n - 2) in
    Pool.wait_until ctx (fun () -> Option.is_some (Atomic.get a));
    Option.get (Atomic.get a) + b
```

Because `Pool.async` does not wait on a result, we use an atomic reference to store the result and wait on it. This pattern is wrapped in an `Ivar` module we contributed to the Zoo standard library, but it is even better to use `Future` directly.

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 practice of library-based schedulers: `Domainslib` requires a `pool` parameter and `Taskflow` requires a `tf::Runtime&` or `tf::Executor&`.

9.3 Specification

The specification is given in Figure 7. It 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-KILL-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-UNTIL-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 `kill` (POOL-KILL-SPEC).

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.

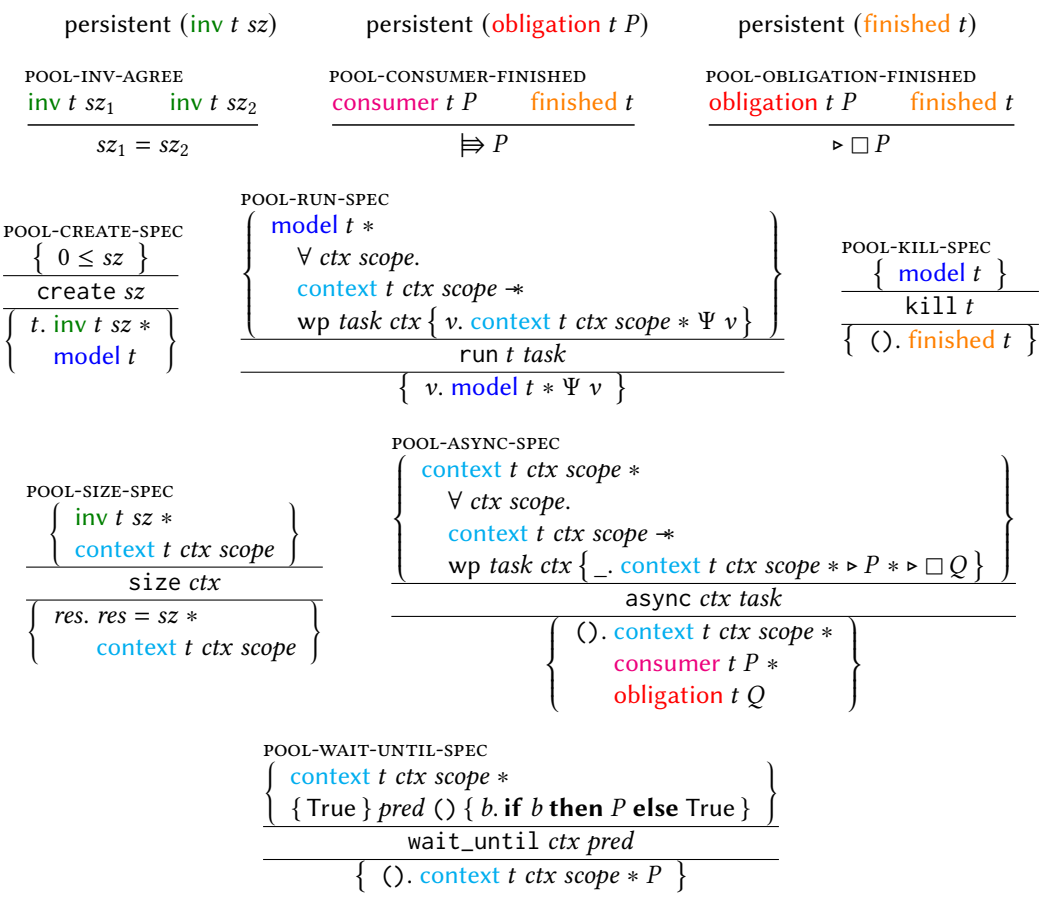


Fig. 7. **Pool**: Specification

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 killed 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 deques properly. When the scheduler is killed, 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_until 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.

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 deques (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 killed.

10 Futures

At the fourth level, **`Future`** implements futures[10], 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.

10.2 Example

We can now write a higher-level version of the `fibo` 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.

[10] 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.

persistent (inv *pool t* Ψ Ξ)

persistent (obligation *pool P*)

persistent (result *t v*)

FUTURE-RESULT-AGREE

result *t* v_1 ∗ result *t* v_2 ⊢ $v_1 = v_2$

FUTURE-INV-RESULT

inv *pool t* Ψ Ξ ∗ result *t v* ⊢ |⇛ ▷ □ Ξ *v*

FUTURE-INV-FINISHED

inv *pool t* Ψ Ξ ∗ pool.finished *pool* ⊢ ▶ ∃ *v*. result *t v*

FUTURE-CONSUMER-DIVIDE

inv *pool t* Ψ Ξ ∗ consumer *t* $\left(\lambda\, v.\ \ast_{X \in Xs}\ X\ v\right)$ ⊢ |⇛ $\ast_{X \in Xs}$ consumer *t* X

FUTURE-INV-RESULT-CONSUMER

inv *pool t* Ψ Ξ ∗ result *t v* ∗ consumer *t* X ⊢ |⇛ ▷² X *v*

FUTURE-OBLIGATION-FINISHED

obligation *pool P* ∗ pool.finished *pool* ⊢ ▶ □ *P*

FUTURE-ASYNC-SPEC

{ pool.context *pool ctx scope* ∗ ∀ *ctx scope*. pool.context *pool ctx scope* −∗ wp *task ctx* { *v*. pool.context *pool ctx scope* ∗ ▷ Ψ *v* ∗ ▷ □ Ξ *v* } } async *ctx task* { *t*. pool.context *pool ctx scope* ∗ inv *pool t* Ψ Ξ ∗ consumer *t* Ψ }

FUTURE-WAIT-SPEC

{ pool.context *pool ctx scope* ∗ inv *pool t* Ψ Ξ } wait *ctx t* { *v*. £ 2 ∗ pool.context *pool ctx scope* ∗ result *t v* }

FUTURE-ITER-SPEC

{ pool.context *pool ctx scope* ∗ inv *pool t* Ψ Ξ ∗ ∀ *ctx scope v*. pool.context *pool ctx scope* −∗ result *t v* −∗ wp *task ctx v* { (). pool.context *pool ctx scope* ∗ ▷ □ *P* } } iter *ctx t task* { (). pool.context *pool ctx scope* ∗ obligation *pool P* }

FUTURE-MAP-SPEC

{ pool.context *pool ctx scope* ∗ inv *pool* t_1 Ψ_1 Ξ_1 ∗ ∀ *ctx scope* v_1. pool.context *pool ctx scope* −∗ result t_1 v_1 −∗ wp *task ctx* v_1 { v_2. pool.context *pool ctx scope* ∗ ▷ Ψ_2 v_2 ∗ ▷ □ Ξ_2 v_2 } } map *ctx* t_1 *task* { t_2. pool.context *pool ctx scope* ∗ inv *pool* t_2 Ψ_2 Ξ_2 ∗ consumer t_2 Ψ_2 }

Fig. 8. **Future**: Specification

`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* fut_1 *task* creates a new future to be resolved after fut_1 (FUTURE-MAP-SPEC).

The persistent assertion inv *pool t* Ψ Ξ represents the knowledge that *t* is a valid future attached to pool *pool* such that: (1) Ψ is the *non-persistent output predicate* satisfied by the produced value; (2) Ξ 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* (concurrent write-once variables), 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_until` 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 ->
  int (* begin index (included) *) ->
  int (* end index (excluded) *) ->
  int option (* recommended sequential cutoff size *) ->
  (int -> int -> unit) Pool.task -> unit
```

and could be used as follows:

```
let double_array ctx t =
  Algo.for_ ctx 0 (Array.length t) 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

$$\textsc{prophet-model-exclusive}\quad \frac{\mathsf{model}\ \mathit{pid}\ \mathit{prophs}_1 \qquad \mathsf{model}\ \mathit{pid}\ \mathit{prophs}_2}{\mathsf{False}}$$

$$\textsc{wp-proph}\quad \frac{\{\ \mathsf{True}\ \}}{\mathsf{Proph}}}{\{\ \mathit{pid}.\ \exists\ \mathit{prophs}.\ \mathsf{model}\ \mathit{pid}\ \mathit{prophs}\ \}}$$

$$\textsc{wp-resolve}\quad \frac{\mathsf{to\text{-}val}\ e = \mathsf{None} \qquad \mathsf{model}\ \mathit{pid}\ \mathit{prophs} \qquad \mathsf{atomic}\ e \qquad \mathsf{wp}\ e \left\{ \begin{array}{l} \mathit{res}.\ \forall\ \mathit{prophs}'. \\ \quad \mathit{prophs} = (\mathit{res}, v) :: \mathit{prophs}' \mathrel{-\!\!*} \\ \quad \mathsf{model}\ \mathit{pid}\ \mathit{prophs}' \mathrel{-\!\!*} \\ \quad \Phi\ \mathit{res} \end{array} \right\}}{\mathsf{wp\ Resolve}\ e\ \mathit{pid}\ v \{\ \Phi\ \}}$$

Fig. 9. Reasoning rules for primitive prophets

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, we do not present our benchmark results here. They can be found in 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.

WP-PROPH says that `Proph` allocates a new prophet with some unknown prophecies to be resolved. WP-RESOLVE says that `Resolve` *e 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 τ along with two functions of-val and to-val.

$$\frac{\text{model } \mathit{pid}\ \mathit{prophs}_1 \qquad \text{model } \mathit{pid}\ \mathit{prophs}_2}{\text{False}}\ \text{TYPED-PROPHET-MODEL-EXCLUSIVE}$$

$$\frac{\dfrac{\{\ \text{True}\ \}}{\text{Proph}}}{\{\ \mathit{pid}.\ \exists\ \mathit{prophs}.\ \text{model}\ \mathit{pid}\ \mathit{prophs}\ \}}\ \text{TYPED-PROPHET-PROPH-SPEC}$$

$$\frac{\text{atomic } e \qquad \text{to-val } e = \text{None} \qquad v = \mathit{prophet}.\text{to-val}\ \mathit{proph} \qquad \text{model}\ \mathit{pid}\ \mathit{prophs} \qquad \text{wp}\ e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}'. \\ \quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \twoheadrightarrow \\ \quad \text{model}\ \mathit{pid}\ \mathit{prophs}' \twoheadrightarrow \\ \quad \Phi\ w \end{array} \right\}}{\text{wp Resolve } e\ \mathit{pid}\ v \{\ \Phi\ \}}\ \text{TYPED-PROPHET-RESOLVE-SPEC}$$

Fig. 10. Reasoning rules for typed prophets

to-val converts an inhabitant of τ 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 τ; 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* γ *past prophs* represents the ownership of the prophet with identifier *pid*; γ is the logical name of the prophet; *past* is the list of prophecies resolved so far; *prophs* is the list of prophecies that must still be resolved.

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

The persistent assertion snapshot γ *past prophs* represents a snapshot of the state of the prophet with name γ 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 γ *lb* represents a lower bound on the non-resolved prophecies of the prophet with name γ. 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.

13.4 Multiplexed prophet

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

persistent (full γ *prophs*) persistent (snapshot γ *past prophs*) persistent (lb γ *lb*)

WISE-PROPHET-MODEL-EXCLUSIVE

$$\frac{\text{model } \mathit{pid}\ \gamma_1\ \mathit{past}_1\ \mathit{prophs}_1 \qquad \text{model } \mathit{pid}\ \gamma_2\ \mathit{past}_2\ \mathit{prophs}_2}{\text{False}}$$

WISE-PROPHET-FULL-GET

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\text{full } \gamma\ (\mathit{past} +\!\!+ \mathit{prophs})}$$

WISE-PROPHET-FULL-VALID

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}_1 \qquad \text{full } \gamma\ \mathit{prophs}_2}{\mathit{prophs}_2 = \mathit{past} +\!\!+ \mathit{prophs}_1}$$

WISE-PROPHET-FULL-AGREE

$$\frac{\text{full } \gamma\ \mathit{prophs}_1 \qquad \text{full } \gamma\ \mathit{prophs}_2}{\mathit{prophs}_1 = \mathit{prophs}_2}$$

WISE-PROPHET-SNAPSHOT-GET

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\text{snapshot } \gamma\ \mathit{past}\ \mathit{prophs}}$$

WISE-PROPHET-SNAPSHOT-VALID

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}_1\ \mathit{prophs}_1 \qquad \text{snapshot } \gamma\ \mathit{past}_2\ \mathit{prophs}_2}{\exists\ \mathit{past}_3.\ \mathit{past}_1 = \mathit{past}_2 +\!\!+ \mathit{past}_3 * \mathit{prophs}_2 = \mathit{past}_3 +\!\!+ \mathit{prophs}_1}$$

WISE-PROPHET-LB-GET

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\text{lb } \gamma\ \mathit{prophs}}$$

WISE-PROPHET-LB-VALID

$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs} \qquad \text{lb } \gamma\ \mathit{lb}}{\exists\ \mathit{past}_1\ \mathit{past}_2.\ \mathit{past} = \mathit{past}_1 +\!\!+ \mathit{past}_2 * \mathit{lb} = \mathit{past}_2 +\!\!+ \mathit{prophs}}$$

WISE-PROPHET-PROPH-SPEC

$$\frac{\{\ \text{True}\ \}}{\{\ \mathit{pid}.\ \exists\ \gamma\ \mathit{prophs}.\ \text{model } \mathit{pid}\ \gamma\ []\ \mathit{prophs}\ \}}\ \text{Proph}$$

WISE-PROPHET-RESOLVE-SPEC

$$\frac{\text{atomic } e \qquad \text{to-val } e = \text{None} \qquad v = \mathit{prophet}.\text{to-val } \mathit{proph} \qquad \text{model } \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs} \qquad \text{wp } e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}'. \\ \quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \mathrel{-\!\!*} \\ \quad \text{model } \mathit{pid}\ \gamma\ (\mathit{past} +\!\!+ [\mathit{proph}])\ \mathit{prophs}' \mathrel{-\!\!*} \\ \quad \Phi\ w \end{array} \right\}}{\text{wp Resolve } e\ \mathit{pid}\ v\ \{\ \Phi\ \}}$$

Fig. 11. Reasoning rules for wise prophets

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

14 Related work

Chase-Lev work-stealing deque. Jung, Lee, Choi, Kim, Park and Kang [2023][11] 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

[11] See also the master thesis of Choi [2023].

$$\text{persistent } (\text{full } \gamma\ i\ \mathit{prophs}) \qquad \text{persistent } (\text{snapshot } \gamma\ i\ \mathit{past}\ \mathit{prophs}) \qquad \text{persistent } (\text{lb } \gamma\ i\ \mathit{lb})$$

WISE-PROPHETS-MODEL-EXCLUSIVE
$$\frac{\text{model } \mathit{pid}\ \gamma_1\ \mathit{pasts}_1\ \mathit{prophss}_1 \qquad \text{model } \mathit{pid}\ \gamma_2\ \mathit{pasts}_2\ \mathit{prophss}_2}{\text{False}}$$

WISE-PROPHETS-FULL-GET
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\text{full } \gamma\ i\ (\mathit{pasts}\ i +\!\!+ \mathit{prophss}\ i)}$$

WISE-PROPHETS-FULL-VALID
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \text{full } \gamma\ i\ \mathit{prophs}}{\mathit{prophs} = \mathit{pasts}\ i +\!\!+ \mathit{prophss}\ i}$$

WISE-PROPHETS-FULL-AGREE
$$\frac{\text{full } \gamma\ i\ \mathit{prophs}_1 \qquad \text{full } \gamma\ i\ \mathit{prophs}_2}{\mathit{prophs}_1 = \mathit{prophs}_2}$$

WISE-PROPHETS-SNAPSHOT-GET
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\text{snapshot } \gamma\ (\mathit{pasts}\ i)\ (\mathit{prophss}\ i)}$$

WISE-PROPHETS-SNAPSHOT-VALID
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \text{snapshot } \gamma\ i\ (\mathit{pasts}\ i)\ (\mathit{prophss}\ i)}{\exists\ \mathit{past}'.\ \mathit{pasts}\ i = \mathit{past} +\!\!+ \mathit{past}' * \mathit{prophs} = \mathit{past}' +\!\!+ \mathit{prophss}\ i}$$

WISE-PROPHETS-LB-GET
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\text{lb } \gamma\ i\ (\mathit{prophss}\ i)}$$

WISE-PROPHETS-LB-VALID
$$\frac{\text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \text{lb } \gamma\ i\ \mathit{lb}}{\exists\ \mathit{past}_1\ \mathit{past}_2.\ \mathit{pasts}\ i = \mathit{past}_1 +\!\!+ \mathit{past}_2 * \mathit{lb} = \mathit{past}_2 +\!\!+ \mathit{prophss}\ i}$$

WISE-PROPHETS-PROPH-SPEC
$$\frac{\{\ \text{True}\ \}}{\{\ \mathit{pid}.\ \exists\ \gamma\ \mathit{prophss}.\ \text{model } \mathit{pid}\ \gamma\ (\lambda\ \_.\ [])\ \mathit{prophss}\ \}}\ \text{Proph}$$

WISE-PROPHETS-RESOLVE-SPEC
$$\frac{\text{atomic } e \qquad \text{to-val } e = \text{None} \qquad v = \mathit{prophet}.\text{to-val}\ \mathit{proph} \qquad \text{model } \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\text{wp } e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}. \\ \quad \mathit{prophss}\ i = \mathit{proph} :: \mathit{prophs} \mathrel{-\!\!*} \\ \quad \text{model } \mathit{pid}\ \gamma\ (\text{alter } (\cdot +\!\!+ [\mathit{proph}])\ i\ \mathit{pasts})\ (\mathit{prophss}\ [i \mapsto \mathit{prophs}]) \mathrel{-\!\!*} \\ \quad \Phi\ w \end{array} \right\}}{\text{wp Resolve } e\ \mathit{pid}\ (i, v)\ \{\ \Phi\ \}}$$

Fig. 12. Reasoning rules for multiplexed prophets

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).

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, Ramananandro 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, Marmar 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].

References

Umut 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

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, 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 Emílio 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 Marmar, 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 Civl 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. Domainslib. https://github.com/ocaml-multicore/domainslib

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

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 deques).

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

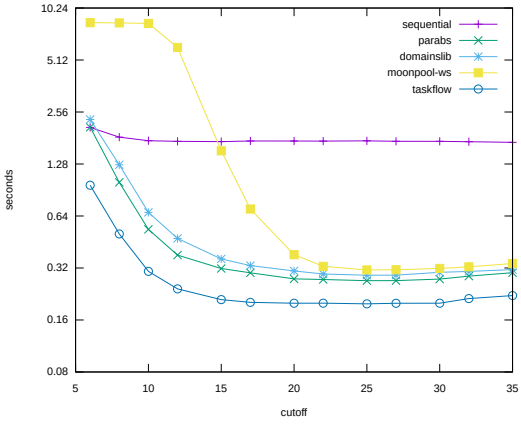
On all schedulers, the performance for larger `CUTOFF` values should be good if the benchmark has homogeneous/regular tasks, and it should be worse if the benchmark has heterogeneous/irregular tasks.

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.

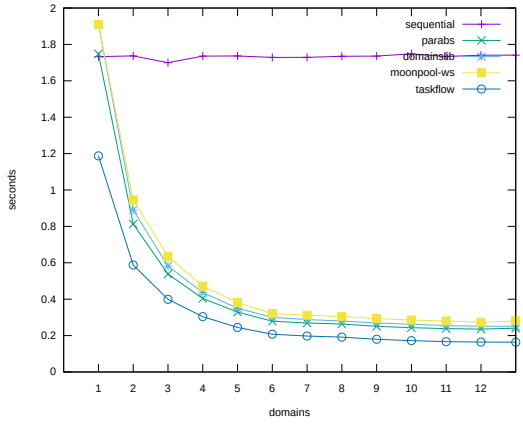
`fibonacci`. As a control, we introduced a `Taskflow` implementation for this benchmark, which is a C++ version using the `Taskflow` scheduler. We expect `Taskflow` to be faster due to a more optimized, industrial implementation and a more aggressively optimizing compiler (`clang++ -O3`).

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.

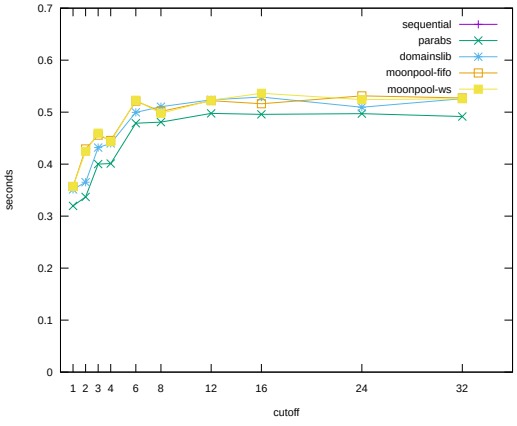
Note: we observe very large memory usage with `Moonpool` at smaller cutoff values — when computing fibonacci 40, attempting to run the benchmark with `CUTOFF` = 5 fails with out-of-memory errors on a machine with 32Gio of RAM. This seems to come from the FIFO architecture which runs the oldest and thus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



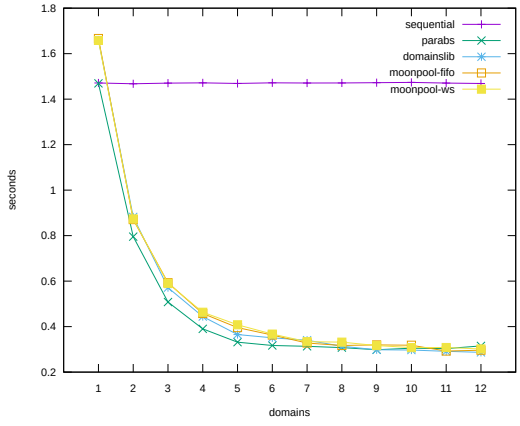
(a) `fibonacci`: varying `CUTOFF`



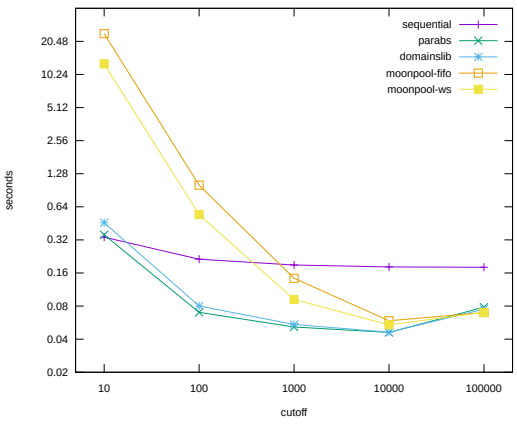
(b) `fibonacci`: varying `DOMAINS`



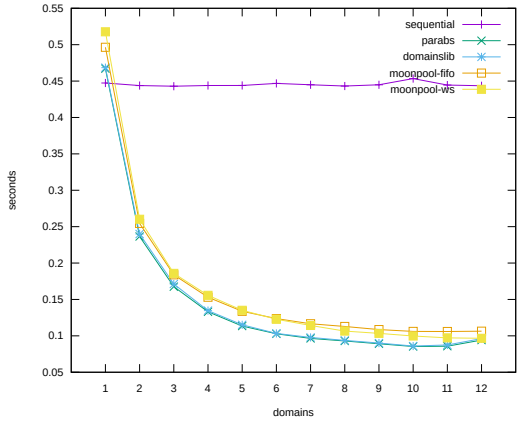
(c) `for_irregular`: varying `CUTOFF`



(d) `for_irregular`: varying `DOMAINS`

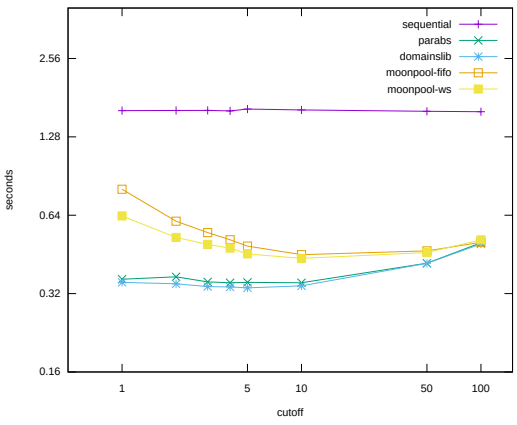


(e) `iota`: varying `CUTOFF`

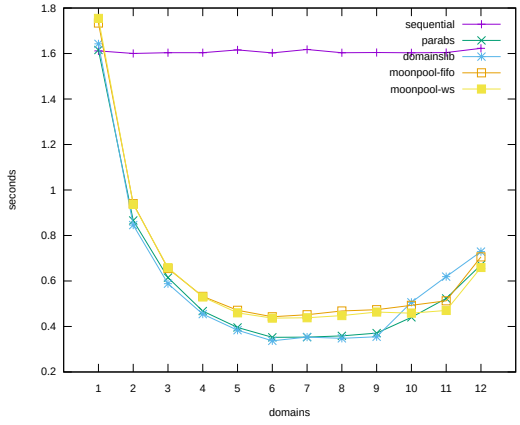


(f) `iota`: varying `DOMAINS`

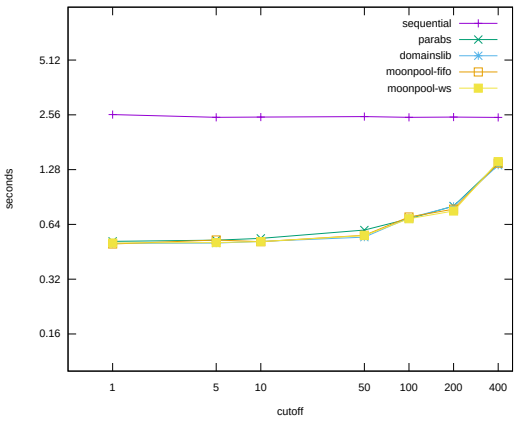
Fig. 13. Benchmarks (1/2)



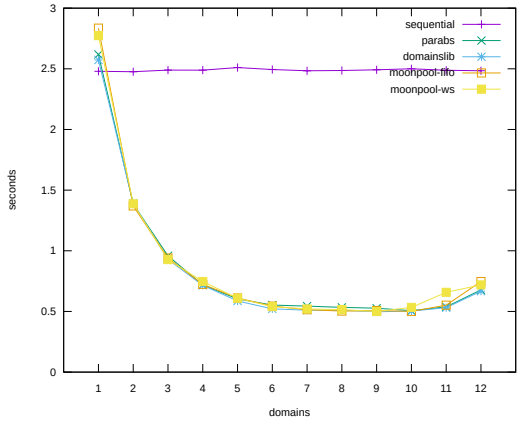
(g) `lu`: varying `CUTOFF`



(h) `lu`: varying `DOMAINS`



(i) `matmul`: varying `CUTOFF`



(j) `matmul`: varying `DOMAINS`

Fig. 13. Benchmarks (2/2)

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 (`DOMAINS` ≥ 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 `DOMAINS` ≥ 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 `DOMAINS` ≥ 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 agressively 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.