

**YaleNUSCollege**

**Concurrent Structures and Effect Handlers:  
A Batch Made in Heaven**

**Lee Koon Wen**

**Capstone Final Report for BSc (Honours) in  
Mathematical, Computational and Statistical Sciences**

**Supervised by: Ilya Sergey**

**AY 2022/2023**

## Yale-NUS College Capstone Project

**DECLARATION & CONSENT**

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property ([Yale-NUS HR 039](#)).

**ACCESS LEVEL**

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

Unrestricted access

Make the Thesis immediately available for worldwide access.

Access restricted to Yale-NUS College for a limited period

Make the Thesis immediately available for Yale-NUS College access only from   03/2023   (mm/yyyy) to   06/2023   (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):

  To avoid conflict with anonymous paper submission made to ICFP on the same topic  

After this period, the Thesis will be made available for worldwide access.

Other restrictions: (please specify if any part of your thesis should be restricted)


---



---

Lee Koon Wen, Saga

Name & Residential College of Student

  
Signature of Student

01/03/2023

Date

Ilya Sergey

Name & Signature of Supervisor



01/04/2023

Date

## *Acknowledgements*

I owe the completion of this project to a team of outstanding people. Thank you to my supervisor, Dr. Ilya Sergey, whose tutelage and guidance have been instrumental in getting me to this point. His ability to bring together amazingly talented people – namely, Dr. Seth Gilbert and Kiran – has made this project more thorough than I could have managed. Above all, he has provided me with opportunity after opportunity, pushing me to be successful in this area.

To Kiran, thank you for your energy, enthusiasm, and wisdom. Your patience and generous counsel have helped me to clarify my doubts and grow my knowledge. Without your hard work, none of this, nor GopCaml, would be possible. Your passion for OCaml ignites my own.

I have also been blessed with the opportunity to surround myself with the people at Tarides and the OCaml community. Of these, I would especially like to mention Arthur Wendling, whose valuable advice pushed me in the direction I needed.

To my parents, thank you for always keeping me well and nourished. You guys never ask for anything other than for me to do my best. To my suitemates and friends – Lucas, Gary, Jon, Gayle, and Claudia – thank you for being staples in this process. For the friendship, laughter, and memories that kept it lighthearted. Finally, to Misaki, thank you for believing in me, for the continuous support, and for seeing me through it all.

YALE-NUS COLLEGE

# *Abstract*

B.Sc (Hons)

## **Concurrent Structures and Effect Handlers: A Batch Made in Heaven**

by Lee KOON WEN

Batch parallelism is a technique for developing highly efficient concurrent data structures. It is based on the observation that processing a batch of *a priori* known operations in parallel is easier than optimizing performance for an arbitrary, asynchronous stream of requests. Despite their simplicity and efficiency, batch-parallel data structures remain unpopular in practice for two reasons. Firstly, their implementation requires careful interleaving of sequential and concurrent code, making them prone to bugs. Secondly, their usage requires users to manually batch data structure operations, making them unnatural to use and challenging to integrate into existing codebases. This paper presents OBATCHER, a Multicore OCaml library that streamlines the design, implementation, and testing of concurrent batch-parallel structures. With OBATCHER, one can take a sequential data structure and easily transform it into its equivalent that supports batch-parallelism and then from there, incrementally retrofit on parallelism. Furthermore, OBATCHER makes batching operations implicit, meaning that the fact that the underlying data structure works on batches is opaque to the user. The benefit of this is that now no change in the program

structure is required whilst users get to enjoy the performance gains of batch-parallelism working under the hood. This paper showcases one case study of how we can take an instance of a set based on a skip list and convert it with `OBATCHER` into an efficient concurrent batch-parallel version. We demonstrate that this implementation outperforms the corresponding coarse-grained lock-based implementations in OCaml and has predictable throughput scaling with the number of processors.

# Contents

<b>Acknowledgements</b>	<b>ii</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Approach . . . . .	3
1.3 Contributions . . . . .	5
<b>2 Overview</b>	<b>6</b>
2.1 Surveying our synchronization options . . . . .	6
2.2 Going Batch-parallel . . . . .	8
2.3 Back to Direct-Style . . . . .	10
2.4 Putting It All Together . . . . .	13
<b>3 Design</b>	<b>16</b>
3.1 A Library for Batch-Parallel Data Structures . . . . .	16
3.1.1 Extended data type . . . . .	18
3.1.2 Worker function . . . . .	18
3.1.3 Direct-Style interface . . . . .	19
3.2 Testing . . . . .	19

<b>4 Case study</b>	<b>22</b>
4.1 Batch-Parallel Skip List . . . . .	22
4.1.1 Sequential skip list overview . . . . .	23
4.1.2 Batch-parallel skip list . . . . .	24
4.1.3 Experiments . . . . .	27
4.1.4 Comparison with a fine-grained skip list . . . . .	28
<b>5 Discussion</b>	<b>30</b>
5.1 Optimal Batch Sizes . . . . .	30
<b>6 Related Work</b>	<b>33</b>
6.1 Continuations, effect handlers, and concurrency . . . . .	33
6.2 Batch parallelism and data parallelism . . . . .	34
6.3 BATCHER . . . . .	35
<b>7 Conclusion</b>	<b>36</b>
<b>Bibliography</b>	<b>37</b>

# 1 Introduction

## 1.1 Motivation

Hardware has evolved over time to give us the multicore architectures we have today. A direct result of that is the ability to speed up our programs by distributing the work between its cores, effectively running work in parallel. A useful abstraction for designing parallel programs that have access to shared memory are data structures that are appropriate for use in concurrent environments, in other words, they are *thread-safe*. Thread-safety means that parallel access to these data structures are guaranteed not to result in an invalid state of the data structure. Unfortunately, designing efficient concurrent data structures that utilize high degrees of parallelism are challenging because of the intricate invariants that must be preserved by possibly overlapping concurrent operations. The two common approaches to convert a sequential data structure into a concurrent version are illustrated next.

The first approach, known as *coarse-grained concurrency*, involves *synchronizing* all operations of the data structure, which is protected by a single lock that must be acquired before any operation is invoked and released afterwards. Although this approach is mostly mechanical for making sequential data structures support concurrency, it removes the opportunity for parallelism *within*



the data structure, making operations mutually exclusive and introduces sequential bottlenecks. As a result, the performance of coarse-grained concurrent structures does not scale linearly with the number of processors due to contention when multiple threads try to access the same memory fragment.

The second approach, *fine-grained concurrency*, requires careful consideration of possible interactions between multiple concurrent operations of a data structure that can overlap in time. Synchronization is introduced sparingly to ensure the correctness of updates and query results, allowing multiple threads to execute the data structure's operations in parallel, improving its throughput. However, this approach significantly increases conceptual complexity and requires non-trivial expertise (Herlihy and Shavit, 2008), as fine-grained concurrent data structures are known to be extremely difficult to design and implement correctly. Their formal verification is still an active research field. (Sergey, Nanevski, and Banerjee, 2015; Feldman et al., 2020; Mulder, Krebbers, and Geuvers, 2022; Meyer, Wies, and Wolff, 2022; Öhman and Nanevski, 2022)

Both these approaches represent two extremes in the trade-off between parallel performance and implementation complexity. However, is there a third option that meets us in the middle? In this paper, we explore *batch-parallelism*, a design pattern slated to offer us this balance. Unlike traditional concurrent data structures that handle arbitrary asynchronous requests one by one, batch-parallel data structures are handed a *batch* of operations to process. Only a single batch runs at a time and these structures can exploit parallelism by dynamically spawning asynchronous computations within the batch. We identify three distinct advantages of such an approach:

1. Greater opportunities for optimization
2. Control over processing order of operations in the batch

### 3. Systematic derivation from sequential data structures and incremental retrofitting of parallelism

Unfortunately, these benefits come at the cost of the program needing to be structured to explicitly supply batches. In practice, this is often impractical and awkward to adapt in common asynchronous contexts. To address this issue, (Agrawal et al., 2014) proposes a technique called implicit batching, which uses a custom scheduler that transparently batches individual requests made by client threads before sending them to a batch-parallel data structure.

Currently, the only existing prototype implementation of implicit batching in Cilk-5 involved modifying its runtime scheduler's internals, making it a heavyweight solution that is difficult to reproduce in most modern programming languages. Additionally, their implementation only allows for one batch-parallel data structure per program instance.

Given the state of the art, we ask the following question:

Can we use existing mechanisms of *higher-order programming* with *composable effects* to implement *efficient, easy-to-understand, and easy-to-use* concurrent data structures via batch-parallelism?

In the rest of this paper, we answer this affirmatively, in OCaml.

## 1.2 Approach

OCaml is a variant of the ML family of programming languages. This year, the language releases its latest version, OCaml 5, also known as Multicore OCaml. This version adds support for writing parallel and concurrent programs via the new *domains* and *effect* features respectively. Domains are OCaml's abstraction

over kernel threads. These heavyweight threads run on separate cores and perform work truly in parallel. Effects and effect handlers on the other hand, are designed to give users the flexibility to build non-local control flow abstractions. They enable, among others, implementations of lightweight threading such as coroutines, green threads, and `async/await`. Our main observation is that the effect handler mechanism, is sufficient to address the two main blockers that prevent batch-parallelism from becoming a mainstream technology for implementing efficient concurrent data structures. Those challenges being:

1. Supporting implicit batch-parallelism without having to modify the underlying runtime scheduler;
2. Accommodating multiple batch-parallel concurrent structures within a single program, and ensuring that computational resources are shared fairly between them.

We address challenge 1 by observing that implicit batching can be emulated in lightweight form without modifying to the system scheduler. We achieve this by requiring the client code of a data structure to provide a *continuation* to the rest of its computation that will be invoked once its request to the data structure has been served. The ability to capture continuations and run them later are the facility that effect handlers provide, allowing us to elegantly implement this solution whilst retaining our *direct style* of programming.

We solve the challenge 2 by leveraging the concept of a *scheduler library* (Dolan et al., 2017; Modelski, 2022; Li et al., 2007). Typically schedulers for lightweight-threading are integrated as part of a programming language's runtime environment. In OCaml, effects give us the ability to now define custom schedulers as a user-level library that wraps our concurrent programs. We design

the implicit-batching mechanism such that it can be transparently integrated on top of these user-level schedulers, relying on its default scheduling policy to distribute the batch-parallel data structure(s) work.

By putting the two together, we are able to decompose the task of designing and integrating batch-parallel data structures into a *lightweight* and *modular* library. With our library, users can expect to design batch-parallel data structures in an incremental fashion and have a seamless way of integrating them into their programs. Furthermore, since our framework decouples the scheduler from the implicit batching mechanism, this enables them to be individually optimized for their specific usecase.

### 1.3 Contributions

Implementing concurrency control with control operators is not a novel concept. Before effect handlers, it was shown that one could implement a task scheduler using first-class continuations (Reppy, 1992; Kiselyov and Shan, 2007). The fact that the same can be done with effect handlers, which are capable of capturing and running one-shot continuations, should come as no surprise. Our key conceptual contributions are thus, (a) the observation that scheduling via effects lifts implicit batching support outside of the runtime. and (b) demonstrating how this can be done in OCaml. Our key practical contribution is OBATCHER, an OCaml library for batch-parallel concurrent structures, which facilitates their implementation and testing while making their usage transparent to the clients. We showcase OBATCHER with a case study of one instance of a concurrent set structure: a skip list (Sec. 4.1), which we have adapted from the literature (Agrawal et al., 2014). Our implementation demonstrates that it outperforms its coarse-grained version under diverse concurrent workloads.

## 2 Overview

In this overview, we illustrate one use case of OBATCHER: migrating a sequential program into a concurrent one. As an example, consider the following simple webserver logic that counts the number of requests that it has served.

```
let c = Counter.init ()
let handle_request = fun _ ->
  Counter.incr c;
  printf "you are the %d'th visitor!" (Counter.get c)
```

Our current program keeps a global instantiation of a counter and has a request handler that increments the counter when invoked. Subsequently, the server retrieves the current count from the data structure and forwards a message to the client, informing them of their visitor number. To safely port this code to run in a concurrent environment, the first order of business is to ensure that our underlying `Counter` module is thread-safe.

### 2.1 Surveying our synchronization options

OCaml programs have traditionally expected to only run under a single-thread of execution. As such, it is entirely safe to have had the `Counter` module be a simple wrapper over a mutable reference as shown in [Fig. 2.1a](#). In a concurrent setting however, this is vulnerable to *data races* where multiple threads

```
1 module Counter = struct
2   type t = int ref
3   let init () = ref 0
4   let incr c =
5     c := !c + 1
6   let get c = !c
7
8 end
```

```
1 module CoarseCounter = struct
2   type t = int ref * Mutex.t
3   let init () = ref 0, Mutex.make ()
4   let incr (c, l) = Mutex.with_lock l
5     (fun () -> c := !c + 1)
6   let get (c, l) = Mutex.with_lock l
7     (fun () -> !c)
8 end
```

(A) Sequential counter

(B) Coarse-grained counter

FIGURE 2.1: A sequential and a coarse-grained lock-based counter.

have modifying access to a piece of shared memory without proper synchronization - potentially leading to unexpected behaviour.

A quick workaround is to employ the *coarse-grained* locking strategy to guard access to shared memory (Fig. 2.1b). Our new `Counter` implementation now includes a `Mutex` that must be acquired before any modifications to the data structure can be performed, ensuring *mutual exclusion*. Whilst this generic approach solves the problem, it leaves us with a data structure without capacity for concurrent operations and having poor scaling behaviour as increasing core count increases contention for the lock. Conversely, designing a counter that uses *fine-grained* synchronization may mitigate our performance concerns but inevitably require a more careful hand to design and reason about. The implementation complexity also becomes exponentially harder as the data structure in question becomes more intricate. Different contexts favor one over the other, but wouldn't it be nice if we had something in between?

Cue *batch-parallelism*, a design pattern that offers us this middleground. In contrast to our *coarse-grained* solution where lock contention is the killer, batch-parallelism solves this by drawing from the architecture of Hendler et al.'s flat-combiner. The idea here is that we can reduce overhead by altering the way

client threads interact with the data structure. In the coarse-grained implementation, each concurrent thread races to claim the lock to run its own operation, spinning if unsuccessful. Alternatively, in the flat-combiner, threads are organized to work collaboratively. Each thread submits its operation to a ledger before trying to grab the lock. If successful, the thread holding the lock becomes responsible for processing all submitted requests sequentially. If not, it waits for the result that another thread will fulfill for it, thereby reducing contention. Crucially, this construction is also what enables dynamic "batches" to form. The advantage of batches is that now we can have prior knowledge of the operations that will be executed, providing opportunities for optimization.

Batch-parallelism takes this one step further by additionally supporting dynamic multi-threading within a batch execution. Together, we retain the benefits of the flat-combiner while now also being able to spawn asynchronous computation. Overall, with batch-parallel data structures, we can already expect to have better scaling behaviour and opportunities for parallelism out of the box compared to the coarse-grained solution. As we will discover in later sections, we will see that it is also mechanical to derive a bare-bones batch-parallel data structure from a sequential data structure using OBATCHER. Subsequently, parallelism can be added in an incremental fashion, making them substantially easier to construct than a full blown fine-grained data structure.

## 2.2 Going Batch-parallel

By convention batch-parallel data structures have a small interface. Typically, they expose the *types* of operations that the data structure can handle and a

main function that implements the batch processing. In OCaml, we have chosen to encode operations as variants, where the constructors represent the operation names and their arguments represent their parameters. Each operation also includes a callback used to provide a convenient way to return the result of the operation to the caller and trigger the rest of the computation that depends on the result. To keep their usage simple, the type of the callback is a function which takes as input the result of the operation and returns `unit`.

```
type op = Get of int -> unit | Incr of unit -> unit
val run_batch: t -> op array -> unit
```

The above signature is an example of how we would declare the interface of a batch-parallel `Counter`. What's left is to provide the implementation of the batch processing logic `run_batch`. In our case, we use a map-reduce style helper function, `parallel_reduce` to process the batch of operations asynchronously. Internally,

```
let run_batch counter batch =
  let v1 = !counter in
  let delta = parallel_reduce
    (function
      | Get kont -> kont v1; 0
      | Incr kont -> kont (); 1
    ) (+) batch in
  counter := !counter + delta
```

FIGURE 2.2: Batch executor

`parallel_reduce` partitions the batch of operations into chunks, maps them into integer deltas that are summed together. The total delta is then added to the overall count, reflecting the state of the counter at the end of the batch process. The mapping function applied to each operation works by running the callback `kont` with some result and then returning the value the operation contributes to the delta. The careful readers might have noticed that `Get` often receives stale values. However, this is not a problem from the perspective of *linearizability* (Herlihy and Wing, 1990) — a consistency model often used as a criterion for concurrent data structures. This model, declares that the state of the data structure is valid as long as there exists some sequential history that



produces the same result. Here, our corresponding sequential ordering for our batch process is that all `Get`'s happens before before all `Incr`'s.

Although what we currently have is sufficient to function as a batch-parallel counter, it suffers from two usability issues. Firstly, users must now write code interfacing with the data structure in callback-oriented style which is notoriously hard to structure. On top of that, using callbacks to return the result of an operation eagerly gives control back to client code. This could amount to a situation where the client code runs indefinitely long, starving the batch processing thread. Secondly, in order to interface with our batch-parallel data structure, client code has to manually collect and launch the batch process — an overall undesirable usability requirement. Acknowledging these challenges, can we do better to give users a direct-style of interacting with batch-parallel data structures? Can we also abstract the work of batching, making their usage even more seamless? Turns out that with *algebraic effects*, we can.

## 2.3 Back to Direct-Style

Algebraic effects offer us a modular way of designing custom control flow abstractions, paving the way for something more ergonomic than the current callback mechanism. Practically, effects work like first-class restartable exceptions by separating the handling of effects from their invocation. The main interface for working with effects is through the `'a Effect` type which represents an effect with the return type `'a`.

```
effect Await: 'a promise -> 'a Effect.t
```

Here we have declared an effect `Await` which expects an abstract type `'a` promise, representing some deferred computation that resolves to `'a`. To evaluate the effect, we use the built-in `perform` keyword to retrieve its result.

```
let await (promise: 'a promise) : 'a = perform (Await promise)
```

Under the hood, when an effect is performed, the computation is suspended and jumps to the nearest enclosing handler exemplified on the right. The syntac-

```
try f () with
| effect Await promise k ->
  let v = (* run promise *) in
  continue k v
```

FIGURE 2.3: `Await`'s effect handler

tic support for effect handling is synonymous to exception handling with the additional continuation object `k` of type `('a, 'b) continuation`. The continuation represents the suspended computation which can be resumed via the `continue` function of type `('a, 'b) continuation -> 'a -> 'b`. we might implement the `Await` effect (Fig. 2.3) such that when the effect is raised, the handler *forces* the deferred computation and then resumes the continuation with the produced value.

Our short description above is particularly relevant as it demonstrates how algebraic effects can be utilized to design readable non-local control flow logic. This feature is especially beneficial when developing various types of concurrency control mechanisms. OCaml currently does not provide any lightweight threading support (*i.e.* `greenthreads`), instead the language provides a thin wrapper over system threads called `Domains`. Users are encouraged instead to use a combination of algebraic effects and domains to build their own thread pool-like mechanisms, with promise-like abstractions to express scheduling of concurrent work across system threads.

The standard implementation of a thread-pool in OCaml is presented in the design of Fig. 2.4. Here, we declare the type `promise` that refers to the

```
1 type 'a promise
2 type 'a callback : 'a -> unit
3 val run_promise : 'a promise -> 'a callback -> unit
4
5 let worker () =
6   while true do
7     let work = next_work () in
8     try work () with
9     | Await pr k ->
10      let callback v = schedule_work (fun () -> continue k v) in
11      schedule_work (fun () -> run_promise pr callback)
12   end
13
14 let resolve pr v = iter (fun callback -> callback v) pr
```

FIGURE 2.4: Worker thread of thread pool

deferred computation that relies on some result. The thread pool maintains some number of domains where each domain runs the `worker` function to continuously draw tasks from the pool and execute them (lines 6-7). The execution of a task is wrapped in a handler which implements the control logic for the deferred computation. If an execution of some work performs the `Await` effect on a promise object, the worker suspends itself and schedules a new task to work on that promise (line 11). When evaluating the promise, a callback is supplied that will schedule the original task once the promise has completed. Optionally, a promise can also be manually set via the `resolve` function (line 14) which will also schedule its associated callback. Given this thread pool abstraction, users can now enjoy expressing concurrency with a direct-style API and leave the scheduling of the underlying promises to the worker.

We are now equipped to walkthrough the core of `OBATCHER` (Fig. 2.5). When a client calls an operations on the batched data structures, it either gets temporarily promoted to become the thread responsible for launching the batch, or it runs the underlying effects mechanism to suspend itself. Note that for our

```

module type S = sig
  type t
  type 'a op

  type wrapped_op =
    Mk of ('a op * ('a -> unit))
  val init: unit -> t
  val run_batch: t -> pool ->
    wrapped_op array -> unit
end

module BatchedCounter : S = struct
  type t = int ref
  type 'a op = Get: int op
    | Incr: unit op
  type wrapped_op =
    Mk of 'a op * ('a -> unit)
  let init () = ref 0
  let run_batch counter pool batch
    = (* use parallel sum *)
end

```

(A) An explicitly-batched interface

(B) An example instantiation

FIGURE 2.6: OBATCHER's interface for explicitly-batched data structures and an instantiation

set-up, we need to use manual resolution (line 8) since the suspended computation relies on the batch processing thread to trigger its callbacks. Using effects to build our promise abstraction, not only do we have a direct-style of expressing concurrency, our callback resolver now does not directly invoke the client code, but reschedules it to prevent returning control to the client code. Finally, we

```

1 let incr t (* : counter and a channel *) =
2   if (* no currently running batch *) then
3     let batch = Chan.collect_all t in
4     run_batch t pool batch
5   else
6     let promise : unit promise =
7       (* create promise *) in
8     Chan.send t (Incr (resolve pr));
9     await promise

```

FIGURE 2.5: Batched increment with a promise

use a channel to implement implicit batching so that interfacing with the explicit batched data structure can be done with atomic requests.

## 2.4 Putting It All Together

OBATCHER conveniently bundles all this functionality together into a small library. We generalize the explicit batched data structure interface into the

module signature shown in [Fig. 2.6a](#). Type `'a op` is used to represent the operation where `wrapped_op` is necessary for tying the return type of the operation to its resolver. Notably, `run_batch` now includes a `pool` parameter which is used internally by the data structure to dynamically spawn asynchronous work.

OBATCHER exposes a functor `Make` which is applied to a module implementation of type `S` (the explicitly batched data structure). Doing so automatically generates an implicitly batched version of

```
module Make : functor (Struct : S) -> sig
  type t
  type 'a op = 'a Struct.op
  val init : pool -> t
  val apply : t -> 'a op -> 'a
end
```

FIGURE 2.7: Functor interface

the data structure that is integrated with the scheduler. The `apply` function is the new entry point for submitting requests to the batched data structure.

Returning to our original example of the counting web server, we can use OBATCHER to easily automate the process of converting the *explicitly* batched counter presented in [Fig. 2.6b](#)

```
module Counter = OBatcher.Make(BatchedCounter)
let pool = (* new thread pool *)
let c = Counter.init pool
let handle_request = fun _ ->
  Counter.apply c Incr;
  printf "you are the %d'th visitor!"
    (Counter.apply c Get)
```

FIGURE 2.8: OBATCHER in action

into an *implicitly* batched counter. Our newly minted data structure now possesses better scaling properties than its coarse-grained counterpart and requires minimal changes to the original code base.

From a another perspective, OBATCHER is also simpler to use than performing a careful full-blown re-implementation of a data structure into its fine-grained form. This is because batching allows us to easily optimize our data-structure based on the high-level properties of its operation. Taking a

more complex data structure like a `Set`. Suppose that our web server would now like to keep track of the number of *unique* visitors.

```
let handle_request = fun req ->
  if not (Set.apply seen_users (Insert (ip_addr req)))
  then Counter.apply c Incr;
  printf "you are the %d'th visitor!" (Counter.apply c Get)
```

With `OBATCHER`, we could wrap the vanilla `Set` module provided in the OCaml standard library and exploit the fact that membership queries do not mutate the data structure and therefore making them parallelizable. Concretely, we could run membership queries first in parallel and then process all remaining operations sequentially. Afterward, more parallelism and optimizations can be retroactively fitted to the data structure, further improving its performance.

## 3 Design

Our artifact for this work, the OBATCHER framework, consists of three main components. First, a library that automates the process of integrating the scheduler and the explicitly batched data structure, enabling its direct-style use as presented informally in [Sec. 2.4](#). Second, a testing harness to check the linearizability of the batch-parallel data structure. Finally, an example implementation of a batch-parallel skip list using our framework. This section will cover the technical implementation of OBATCHER and illustrate how each modular component interacts with one another. The details of the concrete example of the batch-parallel skip list are later discussed in the [Ch. 4](#).

### 3.1 A Library for Batch-Parallel Data Structures

Visually, [Fig. 3.1](#) outlines how OBATCHER connects the user program with the scheduler and batch-parallel data structures. To the left of the diagram, the client application can spawn asynchronous tasks on the thread pool and directly receive their results. If a client now makes requests to the data structure through the OBATCHER interface, behind the scenes, it gets implicitly batched and scheduled to run on the thread pool. The result of the operation is returned to the client at a later time. Separately, the data structure itself has access to dynamically spawning asynchronous computation that returns to itself. The thread pool used in OBATCHER is a fork of the OCaml community's defacto

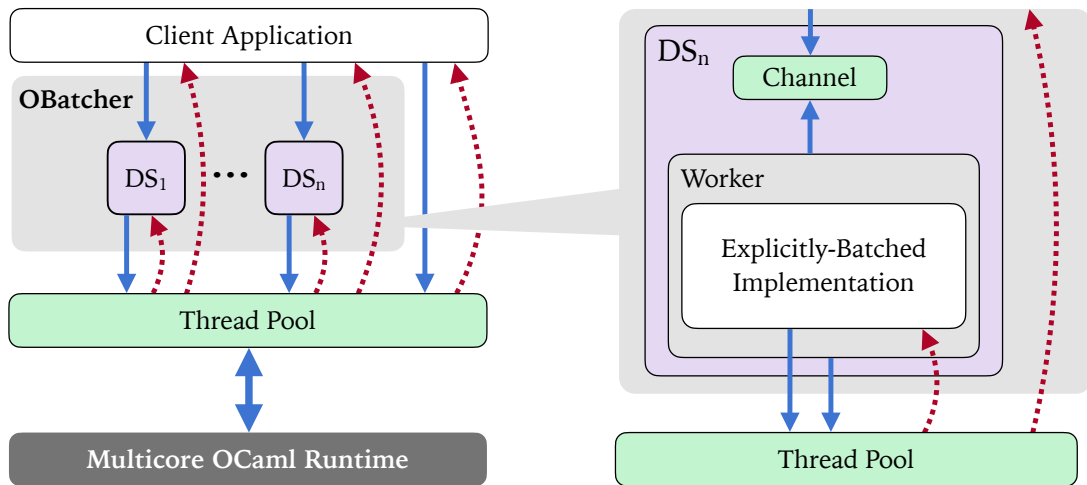


FIGURE 3.1: High-level design of OBATCHER. Boxes with white background contain user-provided components. Light-green boxes are the components that come with default implementations but can be redefined. Purple boxes are data structure instances obtained by instantiating OBATCHER’s functor. Solid blue arrows correspond to direct calls; dashed arrows are callbacks. Interaction between a thread pool and the runtime is not elaborated.

standard `Domainslib`<sup>1</sup>. Importantly, OBATCHER is implemented entirely using a small subset of the API’s exposed by the scheduler. This way, we can keep OBATCHER loosely coupled and thereby easily portable over other schedulers that expose the same interface. The right side of Fig. 3.1 is a more detailed look into how the OBATCHER functor works. For illustration, we deconstruct it into 3 distinct components: (1) The extended type definition of the data structure, (2) The worker function that encapsulates the logic to launch the batch process, and (3) the entry point function that enables the direct-style interface with the data structure. We now address each component individually:

<sup>1</sup><https://github.com/ocaml-multicore/domainslib>, last accessed on 23 February 2023.



### 3.1.1 Extended data type

The `Make` functor provided by `OBATCHER` performs the transformation on a batched data structure by bundling it with a three extra fields. These are specifically, a container

```
type t = {  
  data: (* underlying data type *)  
  container: Chan.t;  
  is_running: bool Atomic.t  
  pool: Task.pool  
}
```

used for storing and retrieving the batch, an atomic boolean used to synchronize the launching of a batch, and a task pool used for dynamically spawning tasks within the batch process. Currently, our implementation uses a channel borrowed from `Domainslib`'s `Chan` module to encode what is essentially a *thread safe* container. Take note that whilst this solution works well, they have been left configurable so that users can customize them to perform more complex preprocessing of the batch that may suit their batch-parallel data structures.

### 3.1.2 Worker function

The worker function encodes the logic of how clients are promoted to become the thread responsible for launching the batch process. We described a simplified version earlier in [Fig. 2.5](#), but more care must be taken to ensure that threads are not starved and no data races are present. When a client invokes the worker function,

```
let rec worker t =  
  if has_requests t.channel &&  
    Atomic.cas t.running false true  
  then begin  
    let batch =  
      Chan.get_all t.channel in  
    run_batch t.data t.pool batch;  
    Atomic.set t.running false;  
    Task.async t.pool  
      (fun () -> worker t)  
  end
```

the thread first checks whether the batch is non-empty and whether another thread is currently the worker. If not, it uses an atomic compare and swap to

try and set the flag before retrieving the batch to run the batch process. To ensure that all operations are eventually picked up and processed, the worker function finally schedules a task on the pool which is a call to itself again. Note that the worker function does not simply recursively call itself to avoid starvation. This can happen if the data structure continuously receives requests, preventing the worker thread from resuming its client code.

### 3.1.3 Direct-Style interface

To expose an idiomatic way of interacting with our data structures, OBATCHER provides the `apply` function that can be used across all supported operations. We are able to

```
let apply t op =  
  let pr, set = Task.promise () in  
  let req = Mk (op, set) in  
  Chan.send t.channel req;  
  worker t;  
  Task.await t.pool pr
```

make this generic by creating empty promises along with their manual resolvers, akin to Java's explicit futures. Next, we submit our request to the channel and then attempt to become the worker. If unsuccessful, we immediately await on our promise which will "under the hood" save its continuation to be executed at a later time when the result is available. From here, the thread can return to helping out with work in the main thread pool.

## 3.2 Testing

While batch-parallel data structures are typically simpler to design than their fine-grained counterparts, this does not remove the fact that they are concurrent algorithms that are hard to reason about. In response, OBATCHER includes an automated testing framework to help users build confidence in the

```
let api = [  
  val_ "insert" Skiplist.insert (fun resolve v1 -> Mk (Insert v1, resolve))  
    (t @-> int @-> returning unit);  
  
  val_ "get" Skiplist.get (fun resolve key -> Mk (Get key, resolve))  
    (t @-> int @-> returning (option int))  
]  
let init () = Skiplist.init ()  
let () = QCheck.run_tests ~count:100 (init,api)
```

FIGURE 3.2: Testing Skiplist Using OBATCHER

correctness of their batch-parallel data structure implementations.

Earlier, in [Sec. 2.2](#), we discussed reasoning about batch-parallel data structures using the linearizability consistency model (Herlihy and Wing, 1990). It turns out that batch-parallel data structures, by design, enable a convenient way to test for linearizability. The testing framework included in OBATCHER builds upon the idea that we can validate if some set of concurrent operations are linearizable by identifying if some sequential ordering exists to explain the results. Specifically, in the case of batch-parallel data structures, the test generates a random set of operations that is submitted to the batched data structure for processing. Subsequently, each result is recorded and upon completion the tool walks through the search space to find a sequential ordering that is able to reproduce the same result. If no sequence is found, an error is raised indicating that the implementation is non-linearizable.

OBATCHER testing harness is adapted from an off-the-shelf linearizability tester, `multicoretests`<sup>2</sup>. To set-up the data structure with the testing suite, the user needs to supply information of the data structure's operations through their DSL described in [Fig. 3.2](#). To declare our batch-parallel skip list for testing, the `api` illustrates the interface that the user needs to provide. (1) The name of the operation `"insert"`, (2) declare the type of the operation and its return

<sup>2</sup><https://github.com/ocaml-multicore/multicoretests>, last accessed on 20 February 2023.

---

value, `t @-> int @-> returning unit`, (3) the reference to the sequential implementation of the operation. `Skiplist.insert` (4) The function that constructs the corresponding reified operation for the interface `Mk (Insert v1, resolve)`. Given this specification, the testing harness will automatically run a series of random tests described previously and check that the data structure's `run_batch` function is linearisable. Furthermore, by hooking into multicoretests's framework, `OBatcher` also gets shrinking of test-cases for free, so when a linearisability violation is found, the counterexample is usually of a reasonable size.

## 4 Case study

### 4.1 Batch-Parallel Skip List

To demonstrate OBATCHER's capability to encode batch-parallel data structures, we implement an efficient concurrent skip list. Even though skip lists were designed by Pugh with concurrency in mind, optimised for data locality and parallelism, turning the original sequential data structure into the one that admits both concurrent search and update operations while reducing contention is not a trivial task. That said, Java's lock-free `ConcurrentSkipListMap` data structure, implemented by Doug Lea using the design by Fraser (2004) is amongst the most efficient and popular container implementations in the `java.util.concurrent` library. The goal of our case study is not to surpass Lea's implementation in its efficiency. Rather, we are aiming to demonstrate that the batch-parallel design allows one to obtain an implementation that exhibits performance trends comparable with those of Fraser-Lea's construction at a fraction of its design complexity. Performance-wise, we observe that with OBATCHER, we manage to reduce contention on the skip list and get good scaling properties: its throughput increases with the number of domains, eventually outperforming its coarse-grained counterpart.

### 4.1.1 Sequential skip list overview

Skip lists are collections of ordered lists that allow for logarithmic-time inserts, search, and deletion. They are often used to implement sets and dictionaries (indices). The power of

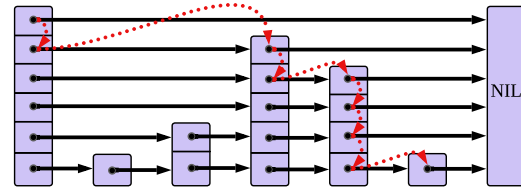


FIGURE 4.1: Searching in a skip list

skip lists lie in their traversal pattern. Unlike vanilla lists that only permit walking through the entire list to reach some node, the skip list algorithm finds a faster path by *skipping* nodes. This is possible thanks to the logical grouping of pointers in skip lists as “levels” whereby each higher level is a sub-list of the level right below it. With this design, the highest level offers the shortest path to the end of the list, the next one “halves” the step distance, *etc.* This provides the structure suitable for a binary search-like traversal pattern — starting from the highest level and moving “right-and-down”.

Skip list is a probabilistic data structure: its implementation features a parameter that controls the randomness determining the highest level that each newly added node appears in it, with *all* nodes appearing at least at the lowest level in the structure. We do not consider probabilistic aspects of skip lists in this work, as they are orthogonal to the concerns of making them concurrent.

**Search** To find a node within the skip list, the search operation begins by traversing the highest level, progressively checking if the key is smaller than the next node. It keeps advancing while the key is smaller, and once it becomes larger than the next node’s value, the search procedure moves the traversal to a more densely populated level below, starting from the last visited node with a smaller key (*cf.* Fig. 4.1). It proceeds by moving down the levels until it either

encounters the sought node or reaches the lowest level. At that point, the node is either located on the last level of traversal, or it is discovered that the desired key is not present in the list.

**Insert** The insert operation works in the same way as the search except that it must keep track of all previous nodes that will have their forward pointers adjusted to the inserted node. Therefore, a node with  $n$  levels where  $n$  is randomly determined, will have to equivalently adjust  $n$  pointers.

### 4.1.2 Batch-parallel skip list

Our batch implementation adopts the skip list implementation in Cilk-5 from the work by Agrawal et al. (2014) on implicit batching and faithfully reproduces it in OCaml by suitably instantiating `OBATCHER`'s functor.

**Search** Search operations, being read-only, are safe to run in parallel with each other in a thread-safe manner, without any need for synchronisation. In our implementation, all search operations in a batch are scheduled to execute and terminate before we start executing insertions.

**Insert** The batch-parallel implementation of insertion for skip-lists runs in three steps: (1) build an intermediary skip list from the batch; (2) perform a parallel search to find the actual slots in the original lists where the new nodes should go, and (3) merge the intermediary list into the original list. Below, we explain all of these steps in detail.

**(1) Intermediary skip list creation** We begin preparing the batch by performing a parallel sort, dropping any duplicates in the process (skip lists implement





To implement this logic, we run a non-mutating parallel search that looks for an appropriate slot *for each new node* in the original list whilst additionally keeping track of the back pointers for all nodes from the original list. As a byproduct of this search, we also obtain forward pointers for each new node based on its discovered location in the original list. In the case where we find a new forward pointer at a corresponding level in the original list, we check if the target node value in the intermediate list is *smaller* than the target value in the original list. If this the case, we know that after merging two lists these links will be preserved, hence no action is required; if not, we perform the necessary remapping from a target node in an intermediary list into the original list.

Consider the example depicted in [Fig. 4.2](#). In our illustration, the lines in red, blue and green represent different threads that are running in parallel to perform the search. The solid lines are pointers that are installed fully and the dotted lines are pointers that are going to be installed. Notice that the new node  $N_1$  to be added has no back pointer ([Fig. 4.2a](#)). Our parallel search-and-update procedure identifies  $O_1$  in the original list is the back pointer for  $N_1$  and updates the table correspondingly. Regarding the forward pointers of  $N_1$ , notice the link  $N_1 \rightarrow N_2$  in the intermediary list should be broken to incorporate the node  $O_2$  from the original list, as shown in [Fig. 4.2b](#).

It crucial that *no modifications* are made to the original skip list during the parallel search and update. If that were not the case, data races in the original list would affect the algorithm's ability to find correct slots. At the same time, it is safe to modify the intermediate skip list because no memory sharing occurs between parallel threads that handle the case of each of the new nodes.

**(3) Merge** The last step is to sequentially walk over the back pointer table and perform the linking into the original skip list. This step can also be performed

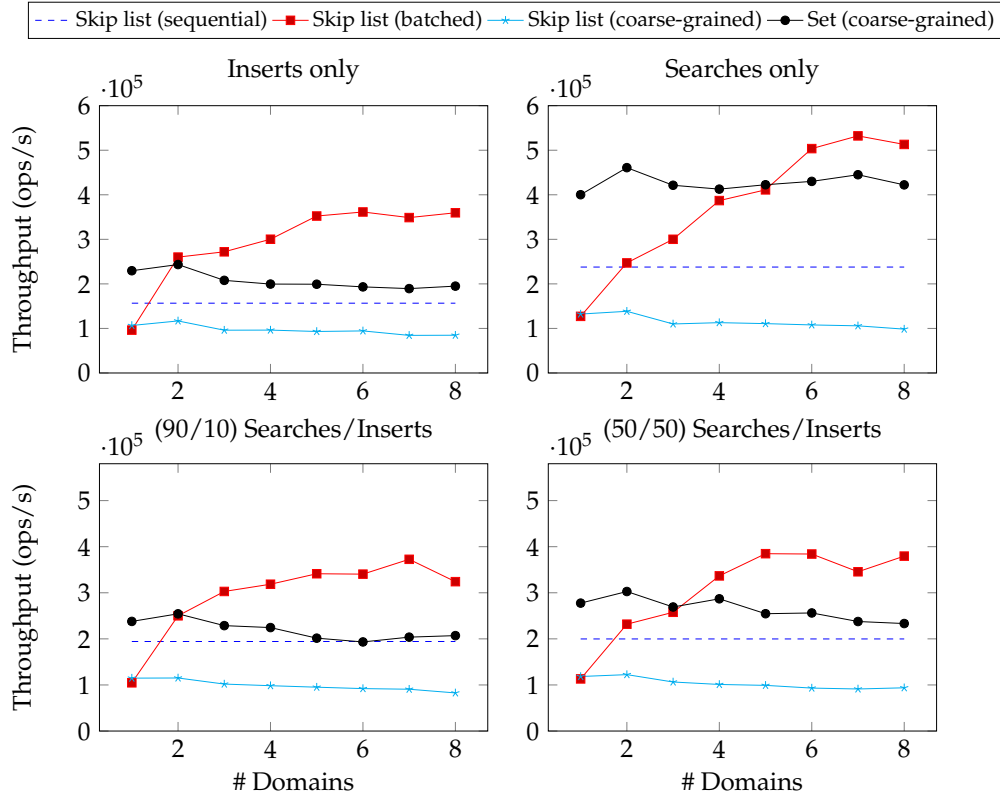


FIGURE 4.3: Comparison of OBATCHER's skip list versus other concurrent data-structures

in parallel since all the correct mappings have been found. However, since this is an inexpensive procedure compared to searching for locations to install nodes, we have left it sequential.

### 4.1.3 Experiments

In our evaluation of OBATCHER, we investigate the effect of varying the number of domains with a fixed 2 million initial elements and a workload size 1 million. We consider four different setups which include inserts only, searches only, 90/10 and 50/50 search/inserts split. Each data point takes the average of 5 runs per configuration. Furthermore, our testing extends to compare the performance of the batched skip list not only with the sequential version, but

also a coarse-grained wrapper over the sequential skip list implemented by us and the OCaml standard library’s `Set` implemented as a self-balancing tree. These changes were made in order to demonstrate how the batched skip list fares against other concurrent skip lists under diverse workloads.

Our findings demonstrate that OBATCHER exhibits good performance with the batched skip list. In all setups, as we increase the number of domains up to the maximal number of available CPUs, lock contention shows performance degradation in both the coarse-grained implementations. Conversely, our batched skip list shows good scaling properties. On diverse workloads, we observe that the batched skip list consistently outperforms the coarse-grained and even the sequential implementation, topping out at about 2x speedup over the sequential with 15 domains. We have run these experiments in OCaml 5.0.0 executed on a PC with a 8-core AMD Ryzen 7 5700G CPU and 32 GB of RAM running on Debian 5.10.136.

#### 4.1.4 Comparison with a fine-grained skip list

Encouraged by the success of the batch-parallel skip list demonstrated by evaluation in Fig. 4.3, we decided to raise the stakes and compare our implementation against a real deal fine-grained skip list—lazy skip list by Herlihy et al. (2007),

whose implementation we have ported

from Java to OCaml. Sadly, one quick look at Fig. 4.4 should be enough to get a clear idea of the growing performance gap between the fine-grained and the batch-parallel implementations. However, instead of simply admitting the

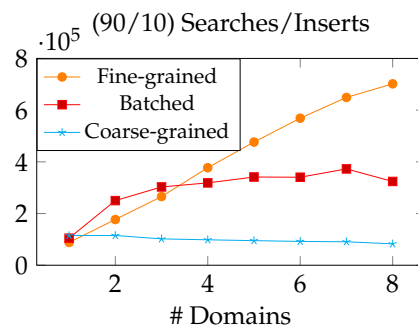


FIGURE 4.4: Fine-grained skip list

crushing defeat by the giants of the multiprocessor programming, we make a few observations that should encourage OCaml programmers not to dismiss OBATCHER as a vehicle for implementing concurrent data structures.

First, while both implementations measure comparable amount of code—200 LOC for the fine-grained skip list and 300 LOC for the batch-parallel one—the fine-grained implementation is significantly more intricate and is much more imperative in its design. Specifically, Herlihy et al.’s skip list makes use of `Atomic` references and introduces locks associated with each node so that the nodes can be physically removed in a “lazy” fashion—the pattern known to be difficult to reason about formally when proving linearisability (Vafeiadis, 2008). In contrast, the implementation of the batch-parallel list does not feature any concurrent programming whatsoever, except for `parallel_for`, which is used for parallelising queries and the search-and-update operation described above. We argue that this approach makes it easier to design useful strategies for parallel processing even for developers who are not experts in fine-grained concurrency, requiring one to reason only in terms of sequential data structure manipulations—as witnessed by our presentation in [Sec. 4.1.2](#).

Second, the batch-parallel design, as presented in this work, is not specific to any particular data structure or even to a class of data structures such as collections. It can be generically applied to any structure using the same approach and even used to migrate sequential libraries to a concurrent setting.

## 5 Discussion

So far, we have poised OBATCHER as a tool that transparently houses the infrastructure which makes the use of batch-parallel data structures practical. That said, it is still however, worthwhile to pay attention to some properties of the design which may inform its use.

### 5.1 Optimal Batch Sizes

One difference between OBATCHER and the BATCHER from Agrawal et al. (2014) paper is the batch size. In BATCHER, the size of the batch is determined by the number of system threads dedicated to running the program. In OBATCHER however, we lift this restriction and make batch sizes configurable by the data structure. By default, the batch size is unbounded allowing it to accommodate a theoretical upper limit of  $n$  operations, where  $n$  is the number of concurrent tasks making requests to the data structure.

A question that naturally arises is: how do we choose an optimal batch size for our batched data structure and program? Setting a large batch size can allow for more operations to accumulate, increasing the gains from parallelism and efficient batch processing. However, users may also preferentially decide to have smaller batch sizes to impose a rate limit, preventing a large batched operation from overwhelming the system. On top of these these two competing interests, it is also not clear how to select an appropriate batch size because

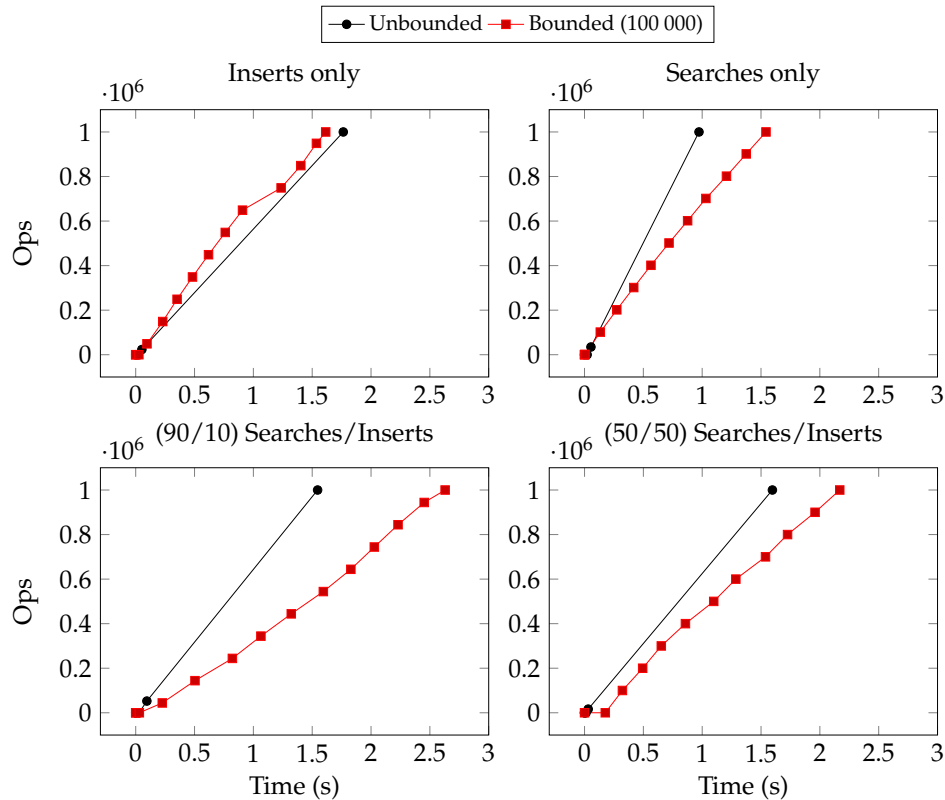


FIGURE 5.1: Batched skip list with varying batch sizes

they depend non-trivially on the program structure and workload. Therefore, finding the best batch size requires some level of trial and error.

To demonstrate this behaviour, we have taken our batched skip list from [Ch. 4](#) and utilized the same benchmarking workloads as before. Instead of varying the number of domains, we fix this value to 8 and now change the batch sizes. To retrieve our results, we keep track of both the dynamic batch sizes along with the time it takes to process them. For each set up, we run 10 iterations and pick the median result of the total time taken for the batch process to complete. Our diagram above shows two lines referring to the case when a batch is unbounded versus bounded with a size of 100,000 elements. In 3 out of 4 cases, unbounded manages to beat the bounded one in terms of total time spent to complete all operations. However, it is important to note that the

cost of an unbounded batch sizes is the latency between batch processes. As depicted in the graphs, the bounded set up has fairly consistent and shorter timings between points whereas the unbounded line demands one large batch and a correspondingly big time slice to complete most of it's operations.

Following this observation, a good metric to keep in mind is: if you don't mind infrequent batch processes demanding a large portion of resources, unbounded is the way to go as you can generally get more out of batch-parallelism with larger batches. However, if responsiveness of the application is a priority, then adding a limit to the batch size can help to break up batch processing into smaller chunks with quicker incremental updates.

## 6 Related Work

### 6.1 Continuations, effect handlers, and concurrency

The idea to use continuations to express concurrency, which effects are based on, has been a well researched topic. The first implementation of a concurrent programs that used continuations was documented by Wand (1980) work that utilized the `call/cc` mechanism in Lisp to implement coroutines. Since then, different iterations of what is fundamentally the same concept has sprouted in different languages such as SML Reppy, 1989; Ramsey, 1990, OCaml Dolan et al., 2017, and Haskell Li et al., 2007. In particular, Reppy’s Concurrent ML draws parallels with our design. Their use of continuations to build a message-passing system with dedicated tasks for each individual data structure is similar to OBATCHER’s worker construction. While Concurrent ML sheds light on some benefits of the design we use in OBATCHER, such as the avoidance of lock contention and the overall simplicity, it does not identify the possibility of batching requests or designing batch-parallel algorithms, which we focus on. From another perspective, Dolan et al. (2017) investigates the use of effect handlers in OCaml for implementing concurrent schedulers. Their body of work emphasizes how to structure concurrent programs, whereas we have concerned ourselves with how to optimize programs that operate on top of these schedulers. Finally, another similar use of continuations is explored by Kiselyov and Shan (2007) to implement a concurrent file system in



Haskell leveraging continuations. Their contribution is a concurrent tree-like data-structure where users requests are represented as suspended continuations similar how requests to batched data-structures are in OBATCHER. Unlike this paper though, that work does not consider batching requests to the data-structure to improve its performance.

## 6.2 Batch parallelism and data parallelism

Data parallelism is a widely used technique in parallel computing. In general, it is applied when there is a large set of data that can be processed independantly. With data parallelism, the same function is applied to different subsets of the data in parallel, with each subset being processed by a separate processor or thread. From this perspective, batch parallelism bears much similarity to data parallelism in that they both deal with batches. The main difference between them is their use case and flexibility. Data parallelism is typically used in the context of applying a *one* aggregate-style bulk operation (*e.g.*, *map*, *reduce*, *filter*, *etc*) to a known collection where each operation runs independantly from each other. In that respect, data parallelism is unaware of other concurrent operations that occur. In contrast, batch parallelism is specifically designed to efficiently handle *multiple* concurrent operations on a data structure while being mindful of potential interference between them. Batch parallelism can extract parallelism from structured data, even if the batch structure is not immediately clear or easy to determine beforehand. Additionally, implicit batch-parallelism can adapt to whatever operations that are performed on the data structure on the fly. In some sense, batch parallelism can be thought of as a superset of data parallelism because batch parallelism

---

can make use of data-parallel techniques. We have demonstrated this with the use of `parallel_reduce` in the batched counter (Fig. 2.2).

### 6.3 BATCHER

OBATCHER is primarily inspired from the work of Agrawal et al.’s BATCHER (2014)—an implicit batching mechanism for batch-parallel data structures in Cilk-5 programs. Instead of trying to be a modular extension of the scheduler, BATCHER adds support for implicit batching by directly baking the logic into the runtime’s randomised work-stealing scheduler. For context, it is important to be aware that the main contribution of Agrawal et al.’s work was to prove certain properties of implicit batching, their aim being to implement a scheduler that had a provable complexity bound for a program that manipulates a batch-parallel data structure. Such a performance theorem would be extremely complicated if it had to consider the interaction between multiple batch-parallel data structures within the same program. In our case, we have decided to focus on a design that prioritizes practicality. In that light, we have opted to feature multiple batch-parallel data structures in the same program at the expense of having any formal performance guarantees.

## 7 Conclusion

In this paper, we have shown how *algebraic effects* and various language support from OCaml has made it possible to provide a *modular* and *lightweight* solution to the design and integration of batch-parallel data structures. We have, as a result of that, created the OBATCHER library to demonstrate its practicality and performance. We have demonstrated through empirical testing that our OBATCHER enables the design of batch-parallel data structures with better scaling and performance than the coarse-grained solution. Furthermore, our tool achieves these results while being overall easier to design than the fine-grained alternative. We hope that with our tool, we have made the usage of batch-parallel data structures accessible, sparking interest in the community toward further development of these efficient concurrent data structures.

# Bibliography

- Agrawal, Kunal, Jeremy T. Fineman, Brendan Sheridan, Jim Sukha, and Robert Utterback (2014). “Provably Good Scheduling for Parallel Programs that Use Data Structures through Implicit Batching”. In: *PPoPP*. ACM, pp. 389–390. DOI: [10.1145/2555243.2555284](https://doi.org/10.1145/2555243.2555284). URL: <https://doi.org/10.1145/2555243.2555284>.
- Dolan, Stephen, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White (2017). “Concurrent System Programming with Effect Handlers”. In: *TFP*. Vol. 10788. LNCS. Springer, pp. 98–117. DOI: [10.1007/978-3-319-89719-6\\_6](https://doi.org/10.1007/978-3-319-89719-6_6).
- Feldman, Yotam M. Y., Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham (2020). “Proving highly-concurrent traversals correct”. In: *Proc. ACM Program. Lang.* 4.OOP-SLA, 128:1–128:29. DOI: [10.1145/3428196](https://doi.org/10.1145/3428196).
- Fraser, Keir (2004). “Practical lock-freedom”. PhD thesis. University of Cambridge.
- Hendler, Danny, Itai Incze, Nir Shavit, and Moran Tzafrir (2010). “Flat Combining and the Synchronization-Parallelism Tradeoff”. In: *SPAA*. ACM, pp. 355–364. DOI: [10.1145/1810479.1810540](https://doi.org/10.1145/1810479.1810540).
- Herlihy, Maurice, Yossi Lev, Victor Luchangco, and Nir Shavit (2007). “A Simple Optimistic Skiplist Algorithm”. In: *SIROCCO*. Vol. 4474. LNCS. Springer, pp. 124–138. DOI: [10.1007/978-3-540-72951-8\\_11](https://doi.org/10.1007/978-3-540-72951-8_11).

- Herlihy, Maurice and Nir Shavit (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann. ISBN: 978-0-12-370591-4.
- Herlihy, Maurice and Jeannette M. Wing (1990). “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3, pp. 463–492. DOI: [10.1145/78969.78972](https://doi.org/10.1145/78969.78972).
- Kiselyov, Oleg and Chung-chieh Shan (2007). “Delimited Continuations in Operating Systems”. In: *6th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT)*. Vol. 4635. LNCS. Springer, pp. 291–302. DOI: [10.1007/978-3-540-74255-5\\_22](https://doi.org/10.1007/978-3-540-74255-5_22). URL: [https://doi.org/10.1007/978-3-540-74255-5\\_22](https://doi.org/10.1007/978-3-540-74255-5_22).
- Li, Peng, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach (2007). “Lightweight concurrency primitives for GHC”. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell*. ACM, pp. 107–118. DOI: [10.1145/1291201.1291217](https://doi.org/10.1145/1291201.1291217). URL: <https://doi.org/10.1145/1291201.1291217>.
- Meyer, Roland, Thomas Wies, and Sebastian Wolff (2022). “A concurrent program logic with a future and history”. In: *Proc. ACM Program. Lang.* 6.OOP-SLA2, pp. 1378–1407. DOI: [10.1145/3563337](https://doi.org/10.1145/3563337). URL: <https://doi.org/10.1145/3563337>.
- Modelska, Bartosz (2022). “Using Effect Handlers for Efficient Parallel Scheduling”. MA thesis. University of Cambridge.
- Mulder, Ike, Robbert Krebbers, and Herman Geuvers (2022). “Diaframe: automated verification of fine-grained concurrent programs in Iris”. In: *PLDI*. ACM, pp. 809–824. DOI: [10.1145/3519939.3523432](https://doi.org/10.1145/3519939.3523432).
- Öhman, Joakim and Aleksandar Nanevski (2022). “Visibility reasoning for concurrent snapshot algorithms”. In: *Proc. ACM Program. Lang.* 6.POPL, pp. 1–30. DOI: [10.1145/3498694](https://doi.org/10.1145/3498694).

- Pugh, William (1990). "Skip Lists: A Probabilistic Alternative to Balanced Trees".  
In: *Communications of the ACM* 33.6, pp. 668–676. DOI: [10 . 1145 / 78973 .  
78977](https://doi.org/10.1145/78973.78977).
- Ramsey, Norman (1990). *Concurrent Programming in ML*. Tech. rep. TR-262-90.  
Princeton University.
- Reppy, John H. (1989). *First-class synchronous operations in Standard ML*. Tech.  
rep. Cornell University.
- (1992). "Higher-Order Concurrency". PhD thesis. Cornell University.
- Sergey, Ilya, Aleksandar Nanevski, and Anindya Banerjee (2015). "Mechanized  
Verification of Fine-Grained Concurrent Programs". In: *PLDI*. ACM, pp. 77–  
87. DOI: [10 . 1145/2737924 . 2737964](https://doi.org/10.1145/2737924.2737964).
- Vafeiadis, Viktor (2008). "Modular fine-grained concurrency verification". PhD  
thesis. University of Cambridge.
- Wand, Mitchell (1980). "Continuation-Based Multiprocessing". In: *Proceedings  
of the 1980 LISP Conference*. ACM, pp. 19–28. DOI: [10 . 1145/800087 . 802786](https://doi.org/10.1145/800087.802786).