Concurrent Data Structures Made Easy

CALLISTA LE, Yale-NUS College, Singapore
KIRAN GOPINATHAN, National University of Singapore, Singapore
KOON WEN LEE, Ahrefs, Singapore
SETH GILBERT, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

Design of an efficient thread-safe concurrent data structure is a balancing act between its implementation complexity and performance. *Lock-based* concurrent data structures, which are relatively easy to derive from their sequential counterparts and to prove thread-safe, suffer from poor throughput under even light multi-threaded workload. At the same time, *lock-free* concurrent structures allow for high throughput, but are notoriously difficult to get right and require careful reasoning to formally establish their correctness.

In this work, we explore a solution to this conundrum based on a relatively old idea of *batch parallelism*—an approach for designing high-throughput concurrent data structures via a simple insight: efficiently processing a batch of *a priori known* operations in parallel is easier than optimising performance for a stream of arbitrary asynchronous requests. Alas, batch-parallel structures have not seen wide practical adoption due to (*i*) the inconvenience of having to structure multi-threaded programs to explicitly group operations and (*ii*) the lack of a systematic methodology to implement batch-parallel structures as simply as lock-based ones.

We present OBatcher—a Multicore OCaml library that streamlines the design, implementation, and usage of batch-parallel structures. It solves the first challenge (how to use) by suggesting a new lightweight *implicit batching* design that is built on top of generic asynchronous programming mechanisms. The second challenge (how to implement) is addressed by identifying *a family of strategies* for converting common sequential structures into efficient batch-parallel ones, and by providing functors that embody those strategies. We showcase OBatcher with a diverse set of benchmarks. Our evaluation of all the implementations on large asynchronous workloads shows that (a) they consistently outperform the corresponding coarse-grained lock-based implementations and that (b) their throughput scales reasonably with the number of processors.

CCS Concepts: • **Computing methodologies → Concurrent algorithms**.

Additional Key Words and Phrases: shared-memory concurrency, batch parallelism, Multicore OCaml

ACM Reference Format:
Callista Le, Kiran Gopinathan, Koon Wen Lee, Seth Gilbert, and Ilya Sergey. 2024. Concurrent Data Structures Made Easy. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 335 (October 2024), 29 pages. https://doi.org/10.1145/3689775

1 Introduction

Mutable concurrent data structures are a key component of multi-threaded programs that run on common shared-memory machines. Efficient concurrent data structures, which allow for high degree of parallelism, are difficult to design and even more difficult to prove correct.

Authors' Contact Information: Callista Le, Yale-NUS College, Singapore, phongnguyen.le@u.yale-nus.edu.sg; Kiran Gopinathan, National University of Singapore, Singapore, mail@kirancodes.me; Koon Wen Lee, Ahrefs, Singapore, koonwen@gmail.com; Seth Gilbert, National University of Singapore, Singapore, seth.gilbert@comp.nus.edu.sg; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2024/10-ART335
https://doi.org/10.1145/3689775

Given a particular sequential data structure, converting it into a thread-safe concurrent counterpart is usually done by following one of the two approaches. In the first approach, all operations of the data structure are made *synchronised*, *i.e.*, protected by a single lock that must be acquired by a thread before invoking any of them and released afterwards. The main advantage of this approach, known as *coarse-grained* concurrency, is that it is easy to implement, making the migration of sequential code to concurrent almost mechanical. The main downside is that it removes any opportunity for parallelism *within* the data structure itself, effectively making all operations with it mutually-exclusive, and, therefore, introducing sequential bottlenecks.

The second approach, dubbed *fine-grained* concurrency, requires one to carefully consider possible interactions between multiple concurrent operations of a data structure that can overlap in time, while being executed by different threads, introducing synchronisation sparingly to ensure the correctness of the updates and query results. The reduced amount of synchronisation allows multiple threads to execute the data structure's operations in parallel, improving its throughput at the cost of significantly increased conceptual complexity: fine-grained concurrent data structures are known to be extremely difficult to design and implement correctly, requiring non-trivial expertise (Herlihy and Shavit 2008), and their formal verification is still an active research field to date (Feldman et al. 2020; Meyer et al. 2022; Mulder et al. 2022; Sergey et al. 2015).

The technique of *flat combining* by Hendler et al. (2010) was an initial attempt to bridge the gap between coarse and fine-grained concurrency. It presented a lock-sharing discipline for coarse-grained data structures that amortised the cost of repeatedly locking and unlocking the structure by instead promoting an arbitrary client to temporarily serve as a dedicated worker that would lock the data structure once and sequentially handle several client requests in a batch. While this strategy was effective at reducing lock contention and thus the overheads of locking, flat combining was not concerned with exploiting parallelism in the data structure itself, leaving the entire construction still substantially worse than a dedicated fine-grained implementation.

Batch-parallel data structures have been proposed as an alternative design pattern that aims to offer a trade-off between parallel performance and the implementation complexity. Batch-parallel data structures differ from traditional concurrent data structures in that they process a *batch* of operations collectively in parallel, instead of handling arbitrary asynchronously incoming requests one by one. Within the batch, such structures exploit parallelism by dynamically spawning asynchronous computations (Acar et al. 2020; Brodal et al. 1998; Dhulipala et al. 2020; Driscoll et al. 1988; Paige and Kruskal 1985). This design comes with multiple advantages:

(1) parallel processing of a batch can be optimised based on the properties of its operations;
(2) the structure can control the order in which the operations in a batch are processed;
(3) batch-parallel data structures can be systematically derived from their sequential versions and incrementally retrofitted with more parallelism.

The main reason why batch-parallel data structures have not been taken up by practitioners is because structuring programs to pass an *explicit batch* of operations is inconvenient and unfeasible in common asynchronous contexts. *Implicit batching* by Agrawal et al. (2014) is a technique meant to circumvent this problem by means of providing a custom scheduler to transparently batch individual requests made by client threads before they are sent to a batch-parallel data structure. The only existing prototype implementation of implicit batching has been done by its authors in Cilk-5 (Frigo et al. 1998) by explicitly modifying its runtime scheduler's internals—hardly a lightweight solution that could be easily reproduced in most modern programming languages. Furthermore, Agrawal et al.'s implementation of implicit batching does not allow to have *more than one* batch-parallel data structure per application, which renders it impractical for real-world tasks.

Given the state of the art, we phrase the motivation for this work as the following question: Can we implement *efficient*, *easy-to-understand*, and *easy-to-use* concurrent data structures via batch parallelism? In the rest of this paper, we answer this question affirmatively.

Key ideas. We observe that the technique of implicit batching can be implemented in a lightweight way *as a library* by re-purposing techniques from flat combining in conjunction with common concurrent programming primitives, namely `async`/`await`, which are available in many modern mainstream programming languages, including Python, Rust, Kotlin, Swift, and, since recently, OCaml. In particular, in OCaml 5 (*a.k.a.* Multicore OCaml), high-level libraries such as Eio and Domainslib implement the primitives for asynchronous programming using the mechanism of *effect handlers* (Sivaramakrishnan et al. 2021). It turns out that the support for asynchronous programming, as provided by Domainslib, is sufficient to address the two shortcomings of Agrawal et al.'s proposal and allows one to (1) implement implicit batch parallelism without modifying the runtime scheduler, and (2) accommodate multiple batch-parallel concurrent data structures within a single program, fairly distributing available computational resources between their operations.

Having attempted to implement batch-parallel versions of common sequential search structures, such as AVL and van Emde Boas trees, we have noticed that many of them fall into one of the two categories when it comes to orchestrating their internal parallelism. In particular, operations of tree-like search structures, whose sub-structures (*i.e.*, subtrees) are themselves valid instances of that data structure, admit a principled parallelisation strategy based on the *split-join* idea by Blelloch et al. (2016) and the concept of bulk updates (Akhremtsev and Sanders 2016; Sanders et al. 2019). In contrast, for search structures that rely on prefixes or hashes to store keys and do not admit a natural "splitting", parallel execution of operations can be done by exploiting the "locality" of their effect—a novel idea that we call *expose-repair*. We capture these observations, as well as a generic structure for batch-parallelism, in an extensible higher-order OCaml library that features functors for the batch parallelism strategies described above and makes it straightforward to implement batch-parallel versions for many common sequential data structures.

Contributions. To summarise, in this work we make the following contributions.

- Our main practical contribution is OBatcher: a Multicore OCaml library that facilitates implementation of batch-parallel structures, while making their usage transparent to the clients (Sec. 3). OBatcher is *lightweight*: it does not require any significant changes in the client code to use it, and implementations in it can be used as drop-in replacements for their coarse-grained analogues. As a framework, OBatcher allows one to develop new batch-parallel structures *gradually* by elaborating the implementation of a function that processes a batch of operations—with the default implementation simply doing so sequentially, with no parallelism whatsoever.
- Our main conceptual contribution is the observation that many search structures, whose concurrent versions are traditionally considered challenging to implement, fall into one of the two categories, so-called "split-join" or "expose-repair", that allow for simple and efficient batch parallelism. We substantiate this observation by providing a family of OCaml functors built on top of OBatcher that abstract away the details of those batching strategies, streamlining implementation of concurrent versions of the respective sequential data structures (Sec. 4).
- We showcase OBatcher by implementing in it a wide range of concurrent search structures, including AVL, Red-Black, and van Emde Boas trees, treaps, x-fast and y-fast tries, B-trees, skip lists, and a concurrent version of a third-party Datalog solver. Our implementations outperform their coarse-grained counterparts in nearly all scenarios under diverse workloads (Sec. 5).

Though we use OCaml as the primary language for our implementation and presentation due to its convenient mechanisms for composition via functors and for concurrency via Domainslib's

```
module Counter = struct
  type t = int ref
  let init () = ref 0
  let incr c =
      c := !c + 1
  let get c = !c

end
```

(a) A sequential counter

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

(b) A coarse-grained counter

Fig. 1. A sequential and a coarse-grained lock-based counter.

primitives, the core ideas of this work generalise beyond a single language and can be easily replicated in any modern language that supports `async`/`await`-style concurrency with multi-producer, multi-consumer channels and task pools, as well as means for code reuse via mechanisms such as type classes or traits. We substantiate this claim with a Rust implementation of the core framework.

2 Overview

We start by building an intuition for OBatcher with a help of a simple concurrent application in OCaml that keeps track of the number of visiting users in a multi-core web-server.

To begin, consider a request handler that outputs the total number of seen visitors of the server:

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

In this program, each time a request is received, the handler calls out to a `Counter` module, first incrementing the count `c` of seen visitors, and then retrieving the current value to print out a welcome message. While this handler could easily be ported from a vanilla OCaml codebase to a multi-core one, the same cannot be said for the underlying `Counter` module that it relies upon.

2.1 A Straw-Man Solution: Coarse-Grained Locking

Classically, OCaml programs have had the benefit of assuming a single-threaded execution, so a vanilla implementation of `Counter` could simply be a wrapper around a reference (*cf.* Fig. 1a).

In the multi-core setting, these operations would not be atomic and introduce a data race in the presence of concurrent calls to `incr` and `get` of the same counter instance. To avoid this, the operations of `Counter` must be written such that they remain correct even in the presence of concurrent executions, for example, by adopting so-called *coarse-grained* locking (Fig. 1b). This would make all manipulations with the counter *mutually-exclusive*, so any concurrent calls to them from different threads would take place sequentially, removing any opportunity for parallelism.

Using coarse-grained locking serves as a general mechanism for migrating existing OCaml code to multi-core, but it leaves an unsatisfying conclusion: we must explicitly rule out any concurrent operations on the underlying data structure. Conversely, constructing an efficient concurrent thread-safe implementation requires careful analysis of how these operations interact, and likely requires entirely redesigning the original structure to introduce *fine-grained* (*i.e.*, internal) synchronisation. Of course, for this pedagogical example, it would be simple to rewrite it to use an atomic variable, but this can not be said of most data structures that might need be ported to a concurrent setting.

Is it possible to find a middle ground between these two extremes, and allow developers to utilise parallelism without having to fully give up on their sequential implementations? With OBatcher, we answer this question in the affirmative, using the idea of operation *batching* to bridge the gap.

2.2 A Batched Interface for Counter

How can we exploit parallelism in the implementation of `Counter`? Unfortunately our current interface where operations are processed individually does not leave much room for optimisation, so let us now consider a different architecture, inspired by Hendler et al.'s flat combiner.

Suppose our interface to `Counter` instead took in *batches* of operations to be executed in parallel:

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

Here, each reified operation `op` includes a callback to allow the return value to be communicated to the caller. To avoid excessive type parameterisations, all callbacks in `op` have a return type `unit`, with the intuition that they are effectful functions—as is the case, *e.g.*, with our counting server.

The signatures above describe a new batched interface for the data structure, which now admits a parallel implementation as presented in Fig. 2. This new implementation uses the helper function `parallel_reduce` to perform a map-reduce style parallel computation. Internally, `parallel_reduce` partitions the operation array into chunks, which are then each mapped to integer deltas in parallel and then summed together to produce the total delta to be applied to the counter. As each submitted operation is processed, the mapping function also invokes callback `kont` with an appropriate return value, informing the client of the corresponding operation's completion.

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

Fig. 2. Batch executor

The careful reader may be wondering about the correctness of this implementation of `run_batch`. In particular, observe that operations may sometimes produce seemingly stale values as their results, such as the value `vl` passed to `Get`'s callback. Fortunately, this behaviour is correct, as it is justified from the perspective of *linearisability* (Herlihy and Wing 1990)—a commonly adopted correctness criterion for concurrent data structures. In short, such "stale" effects of the `Get` operations from a batch in the presence of concurrent interference correspond to a collection of concurrent calls to the `get` method, which happen to take effect *right before* any increment operations taking place at the same time.[1]

Overall, this new implementation provides increased throughput as compared to the coarse-grained wrapper, but comes at the cost of placing more restrictions on the caller, making it challenging to integrate into existing code. Firstly, in contrast to our previous direct-style interface where requests are processed immediately, this new batched API expects a *collection* of requests to be submitted all at once, using callbacks to return their results. Furthermore, from a practical standpoint, the `kont` callbacks should not be continuations of the client's code, as that would return control from the batcher and the thread would be busy computing the client's logic rather than processing other operations in the batch. As it turns out, it is possible to elegantly solve these two problems using the interface provided by Multicore OCaml and Domainslib.

2.3 Implicit Batching via Asynchronous Programming

Let us solve the two shortcomings of the explicitly-batched counter implementation described in Sec. 2.2: (1) the need to explicitly provide a collection of operations and the continuation and (2) the blocking invocation of the user's continuation `kont` within the code `run_batch` in Fig. 2.

[1] Or, in the concurrent programming jargon, these calls *get linearised* before any concurrent increments.

```
1  let incr t (* counter & channel *) : unit =
2    let promise: unit promise,
3        resolve: unit -> unit =
4      Task.promise () in
5    Chan.send t.chan (Incr resolve);
6    if (* no currently running batch *) then
7      let batch : op array =
8           Chan.collect_all t.chan in
9      run_batch t.data pool batch;
10   await promise
```

Fig. 3. Batched increment with a promise

The core design of OBatcher's construction is presented in Fig. 3. When a client calls an operation on the batched data structure in a direct style (*e.g.*, incrementing the counter via `incr`), OBatcher first uses Domainslib to allocate a promise—a concurrent cell primitive that can be used to store a result asynchronously—using `Task.promise` (lines 2-4); this call returns the promise itself and a function that can be called to resolve its value at a later point. The code then enqueues an `Incr` object on a channel associated to this instance, `t.chan`, passing in the resolution function as an argument. OBatcher then checks whether a batch is currently running, and if not promotes the current client to immediately execute the batch of accummulated operations for the data structure (lines 6-9). Finally, OBatcher completes the direct-style interface using the control inversion provided by Domainslib mechanisms to *suspend* the current client task, via the `await` operation (line 10). Intuitively, by awaiting on this newly allocated promise, the continuation of the `incr` operation caller is effectively suspended until the `resolve` function passed to `Incr` is called. This indirection through the promise object means that implementation of Domainslib task scheduling will *not* invoke the continuation directly where it is called (as is done, *e.g.*, in Fig. 2) but will instead reschedule its resumption as a *separate task* in a shared thread pool, thefore, avoiding the blocking issue with the `run_batch` implementation.

2.4 Putting It All Together

Let us now return back to our original example, and see how we can instantiate OBatcher to use an explicitly-batched `Counter` with our web server's request handler.

OBatcher provides the module signature shown in Fig. 4a to describe explicitly-batched structures. This signature is a generalisation of the explicitly-batched interface to `Counter` seen previously. It uses the type `'a op` to represent operations returning results of type `'a`, with `wr_op` being used to tie each operation with a callback to be called on completion. Requests to the data structure are then sent through `run_batch`, which takes in batches of operations as before and executes them. A notable change to the original interface is the additional `pool` parameter to `run_batch`, which is now provided so that the implementation can also schedule tasks to be run on the thread pool.

```
module type Batched = sig
  type t
  type 'a op

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

(a) An explicitly-batched interface

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

(b) An example instantiation

Fig. 4. OBatcher's interface for explicitly-batched data structures and an instantiation

Given a module **S** implementing the explicitly-batched signature **Batched**, OBatcher provides a functor **Make** that can construct a direct-style API to the data structure (Fig. 5). Clients of the data structure can then use the apply operation to send operations to it, and the direct-style interface means that client code can be written as if the results are returned immediately. Internally, when apply is called, OBatcher suspends the client code via await, and uses channels to implicitly batch requests to pass to the run_batch function, as hinted in Sec. 2.3 and will be detailed in Sec. 3.1.2.

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

Fig. 5. A functor for direct-style structure

Fig. 4b presents the straightforward instantiation of OBatcher's signature for our explicitly-batched implementation of **Counter**. Applying OBatcher's **Make** functor to this module then returns a direct-style API, which we can integrate back into our original request handler with minimal changes (Fig. 6). This new implementation, while providing the same API as the initial version, is able to scale more gracefully with the number of concurrent requests, making it a more apropos choice for a web-server.

```
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)
```

Fig. 6. Using a direct-style batched counter

In this way, OBatcher strikes a balance between a heavy-handed coarse-grained locking around existing sequential code and fully concurrent thread-safe re-implementations. For more complex data structures, users can use OBatcher to gradually improve the performance of code by incrementally exploiting more and more high-level properties of the functions it exposes.

For example, suppose we wanted to extend our endpoint to only record *unique* visitors using a set as shown in Fig. 7. An explicitly-batched implementation of a concurrent set could initially be written as a wrapper over a vanilla **Set**, using the fact that membership queries are *pure* to first handle all such queries in a batch in parallel, and then sequentially process all remaining requests. From here, further improvements could be made by a more careful analysis of the data structure, and implementing tailored batch-parallel operations.

```
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)
```

Fig. 7. Counting unique visitors

3 Implementing OBatcher

In this section, we describe the technical details of OBatcher's implementation in Multicore OCaml using asynchronous programming primitives. Then, observing that the design of the embedding is only dependent on async/await functions, we demonstrate how the framework can be ported to other languages by walking through and comparing to a Rust implementation as an example.

3.1 OBatcher in Multicore OCaml

The core of the OBatcher framework is the **Make** functor from Fig. 5, which is provided to the user to create their own instances of concurrent data structures with a familiar direct-style API, from explicitly-batched implementations that follow the **Batched** signature—as we saw with the **Counter** example in Sec. 2.4. Internally, this functor, when instantiated, produces the following three components for each explicitly batch-parallel data structure: (1) an extended type definition with a container for assembling incoming asynchronous operations into batches, (2) a direct-style function apply that acts as the main interface for clients, and (3) a try_launch function that encodes the logic of how batches are collected and forwarded to the underlying implementation via asynchronous programming. Below, we zoom in on the implementations of each one of the components (1)–(3).

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

3.1.1 Extended Data Type. The Make functor from Fig. 5 extends the input data structure's representation data with additional components: a container to collect the incoming requests for operations, an atomic Boolean value is_running to indicate whether a batch is being processed, a reference to the available thread pool to submit the tasks for parallel execution, and the time of the last batch execution. This construction, where each instance of the data structure is given a separate container to batch requests, is fundamentally what allows OBatcher to have *multiple* batched data structures in the same system, adressing one of the main shortcomings of the original design of implicit batching by Agrawal et al. (2014), discussed in Sec. 1. Any thread-safe channel with queue and dequeue functionality can be used as a container for the batched operations, and Domainslib does provide such a channel implementation. However, we note that we do not need the elements inside to be ordered, nor do we ever need to dequeue anything less than all elements. Hence, we implement a simple variation of a thread-safe lock-free stack (Treiber 1986) to use as container that strips away all but two operations: push, which appends atomically to a list, and pop_all, which can be implemented with a simple atomic exchange operation.

```
let apply s op =
  let pr, set = Task.promise () in
  let req = Mk (op, set) in
  Container.send s.container req;
  try_launch s;
  Task.await s.pool pr
```

3.1.2 Direct-Style Interface. The second component of the functor's construction is the function apply, which is exposed as the main entry point for concurrent client interactions with the data structure. This function takes as its arguments an instance s (of the extended type t) of the data structure and the operation's metadata op. Internally, the apply function is implemented following the intuition presented in Sec. 2.3 adapted for the conventions of Domainslib. The function first allocates an empty promise to capture the result of the operation and acquires its resolution function; it then adds the operation metadata and the resolution to the container. Next, the function calls try_launch, potentially running the batch itself, after which the function finally awaits on the initial promise. We elaborate on try_launch below.

```
let rec try_launch s =
  if has_no_requests t.container then () else
  let time = current_sys_time () in
  if not_enough_requests s.container &&
     time -. s.last_run < wait_threshold
  then Task.async s.pool (fun () -> try_launch s)
  else if Atomic.cas t.running false true
  then begin
    let batch =
      Container.get_all s.container in
    s.last_run <- time;
    run_batch s.data s.pool batch;
    Atomic.set s.running false;
    Task.async s.pool (fun () -> try_launch s)
  end
```

Fig. 8. The try_launch function

3.1.3 Launch Function. The final component of the functor is a helper function to capture the logic of client tasks being temporarily promoted to workers and handling batches of requests. We described a simplified representation of this logic earlier in Sec. 2.3, and we now present a more detailed, thread-safe implementation that ensures a minimum batch size for better performance. Client tasks can be promoted to workers whenever they observe that there are enough pending requests and that no other worker is currently running. To avoid a situation where a small number of pending requests are never run, we also set a maximum duration that a batched data structure can remain idle after the last batch run. Should the minimum amount of pending requests not be met and not enough time has passed since the last run, we do not simply exit or spin, but instead we schedule another task to try launching the batch again later. This ensures that we will not be deadlocked by a small number of operations not being serviced and that the current thread will not waste spin cycles waiting either. Note that this design decision is optimised primarily for a highly-concurrent worflkow where batches will accumulate quickly *i.e.* a web server; in a sequential setting, this particular encoding enforces that each operation will take at least as long as the timeout, and a user might instead consider an implementation such as one to be presented in the next subsection without minimum batch sizes. We also schedule another `try_launch` task at the end of each batch run, to avoid another deadlock situation where other clients submit requests after the existing worker has started running a batch.

3.2 Beyond OCaml: OBatcher in Rust

Though the narrative so far has only considered an OCaml implementation of OBatcher, the reader may have noticed that the core of this embedding does not require any language-specific features and only depends on a select few asynchronous programming primitives. Indeed, this captures one of the strengths of the framework: it is *lightweight* and can easily be generalised to other languages. In the rest of this section, we substantiate this claim by presenting a Rust implementation of the OBatcher framework and discuss any significant changes that were needed.

3.2.1 Explicitly Batched Interface. Recall that the main client interface to OBatcher in the OCaml implementation is its signature to describe explicitly-batched data-structures. While Rust does not support functors, we can recreate the essence of this design using its trait mechanism.

```
trait BOp { type Res; }

struct WrOp<Op : BOp>(Op,
   Box<dyn FnOnce(Op::Res) -> ()>);

pub trait Batched {
    type Op : BOp;

    fn init() -> Self;
    async fn run_batch(&mut self,
        ops: Vec<WrOp<Self::Op>>)
      -> ();
}
```

Fig. 9. Explicitly-batched interface

Fig. 9 presents the embedding of OBatcher's explicitly-batched interface into Rust. The core of this encoding is captured in the `Batched` trait, which is almost verbatim ported from the OCaml implementation. In order to implement this trait for a given data-structure, the user is required to supply three components: (1) a type, `Op`, to describe the operations supported by the data structure, (2) a function `init` to create empty instances of the structure and finally (3) an *asynchronous* function `run_batch` which executes a batch of operations on the structure and may exploit parallelism to optimise performance. Note that as Rust's type system has no support for GADTs, we adopt a more convoluted representation of operations and their return values in this new encoding. In particular, operations are constrained through a trait `BOp` with an associated type `Res` that captures the return type of all operations. This is a less precise encoding than before, as now the `Res` type must capture the all possible return types of all operations, so clients must perform additional case analysis to retrieve a usable result. For example, to encode operations for the Counter from Sec. 2, we can define an enumeration type for operations, `enum CounterOp { Get, Incr }`, and instantiate the `BOp` trait with optional integer as the result type: `impl BOp for CounterOp { type Res = Option<i32> }`.

3.2.2 Extended Data Type. The next main component of the OBatcher implementation is the extended data type definition that stores the additional metadata required to implement batching.

The snippet on the right presents how this extended data-type definition can be encoded in Rust. The translation here from OCaml again is fairly mechanical. We define a new struct `BatcherInner`, parameterised by an explicitly batched datatype `B`. This struct maintains 4 pieces of state: (1) an instance of the data structure, (2) a channel to receive operations on, (3) a channel to send operations on, and (4) a boolean to track whether any batch is currently running.

Finally, we wrap this inner data structure behind an atomic reference-counted cell `Arc` to produce the final batched structure `Batcher<B>`. The most significant difference from the OCaml encoding arises from Rust's more precise tracking of references and mutation, which requires us to be more explicit about when references can copied (via `Arc`) and what data is protected by locks (via `Mutex`). Note that for simplicity we use separate locks for the structure and channel; one could wrap both behind one lock to improve performance.

```
struct BatcherInner<B : Batched> {
  data: Mutex<B>,
  recv: Mutex<Receiver<WrOp<B::Op>>>,
  send: Sender<WrOp<B::Op>>,
  is_running: AtomicBool
}

pub struct Batcher<B: Batched>(
    Arc<BatcherInner<B>>);
```

3.2.3 *Direct-Style Interface.*

Having constructed this extended data type, we can now again recreate a direct-style interface to our explicitly-batched data structures by exploiting the `async`/`await` primitives provided by the language to perform control inversion and reuse the same generic pattern as used in our OCaml implementation. The code to do this is presented in the snippet on the right.

```
async fn apply(&self,
    op : B::Op) -> B::Op::Res {
  let (promise, set) = Promise::new();
  let wr_op = WrOp(op, Box::new(set));
  self.0.send.send(wr_op).unwrap();
  let s = self.clone();
  tokio::spawn(async move {
    s.try_launch().await });
  promise.await }
```

This implementation is largely a verbatim translation of the OCaml code (*cf.* Sec. 3.1.2). In this case, the main changes that had to be made were to account for slight discrepancies between the respective languages in their treatment of asynchronous tasks and memory management. In particular, asynchronous tasks in Rust are lazy, and will only be evaluated when forced. As such, in this implementation, when a client attempts to run a batch of requests, it spawns a task (using `tokio::spawn`) which then eagerly runs `try_launch` (to be discussed in the next section) by `await`ing on it. In conjunction to this, Rust's lack of a garbage collector now means that the code must be explicit about when and where references are shared, and in this case, an explicit *clone* of the reference to the batched data-structure is constructed and *moved* into the spawned task to use.

3.2.4 *Launch Function.*

The final component required to complete this re-implementation of OBatcher in Rust is an encoding of the `try_launch` operation which will implement the logic of conditionally promoting clients to workers and collecting and handling batches of requests. Fig. 10 presents the corresponding Rust implementation of the `try_launch` function. The core logic of this function is once again the same as in the OCaml; for simplicity, this implementation omits any minimum requirements on batch sizes and thus the associated timeout mechanism they would require for liveness, but incorporating these would be fairly trivial. The main differences in the implementation arise due to subtleties of Rust's treatment of asynchronous tasks. In particular, Rust internally compiles asynchronous code into finite state machines to optimise their execution.

```
fn try_launch(&self) -> BoxFuture<'static, ()> {
  let s = self.clone();
  async {
    if let Ok(_) = s.0.is_running
          .compare_exchange(false, true) {
      let mut recv = s.0.recv.lock().await;
      let mut ops = vec![];
      if recv.recv_many(&mut ops).await > 0 {
        let mut data = s.0.data.lock().await;
        data.run_batch(ops).await;
      }
      drop(recv);
      s.0.is_running.store(false);
      tokio::spawn(async move {
        s.try_launch().await });
    }
  }.boxed() }
```

Fig. 10. The `try_launch` function

As such, the compiler will reject an attempt to directly call `try_launch` recursively as this induces infinite states in the state machine, and so we must introduce a level of indirection and explicitly allocate the future on the heap by boxing it.

Putting all together, as we have seen from this reimplementation in Rust, the core of OBatcher's design is largely agnostic to the choice of language, and the implementation can be fairly easily ported to any language with suitable asynchronous programming primitives. Generally speaking, we can expect that small changes may be needed to account for slight discrepancies in the specific semantics of asynchronous operations between languages, but otherwise, OBatcher provides a general mechanism for introducing implicit batching into new languages in a lightweight fashion.

4 Engineering Batch-Parallel Implementations

As demonstrated in Sec. 2, OBatcher reduces the challenge of implementing a concurrent data structure to implementing an *explicitly batch-parallel* one, which takes a collection of operations and executes them in a way that maximises parallelism. We argue that this design simplifies the task of engineering thread-safe concurrency, relieving the developer from the need to reason about concurrent interference, atomicity, and invariants, as long as they can argue that the subsets of operations from the batch provided to `run_batch` are safe to run in parallel—which only requires understanding *sequential* behaviour of those operations. The actual concurrent implementation is obtained by instantiating the explicitly-batched interface **`Batched`** from Fig. 4a with the type of the data structure `t` and its operations `op`, constructor `init`, and, most importantly, the function `run_batch` for the parallel batch execution; the rest is handled by the **`Make`** functor from Fig. 5.

One can easily obtain a trivially correct batch-parallel implementation from the sequential one by making `run_batch` execute all given operations *sequentially*, without any parallelism whatsoever. Such an *ad-hoc* solution (*i.e.*, the one that implements `run_batch` explicitly) can be later refined into a "more parallel" one, *e.g.*, by separating all pure operations, to run them in parallel, from the effectful ones, executed sequentially. This highlights one of the main advantages of our methodology: batch-parallel versions can be systemically derived from the original sequential data structure.

One of this work's key observations is that, for certain classes of sequential search structures, it is possible to identify implementation strategies that further streamline the development of their *efficient* batch-parallel counterparts. This section aims to showcase two such strategies, embodied by OCaml functors: so-called *split-join* (Sec. 4.1) and *expose-repair* (Sec. 4.2). In Sec. 4.3 we present several ad-hoc batch-parallel implementations that do not immediately fall under any of the above two categories. We note that the case studies we describe here are limited to search data structures that implement maps and ordered sets (with the exception of Sec. 4.3.2). We do not claim that our implementations are optimal in the theoretical sense, and leave such proofs for the future work.

4.1 Split-Join Batching Strategy

The first batch parallelism strategy we explore is based on the idea of *splitting/joining* by Blelloch et al. (2016). It is effective for various kinds of search structures represented as *balanced binary trees*, such as AVL trees, red-black trees, and treaps, that allow one to divide them into multiple non-connected trees that are themselves valid instances of that tree type. We can then perform insertion/deletion operations on each of these sub-divided trees independently and in parallel, before joining them together again to get the final tree after all operations have been performed. This idea has been previously explored in the works by Akhremtsev and Sanders (2016) and by Sanders et al. (2019), termed "bulk updates" or "bulk operations", albeit not with the goal to derive efficient concurrent data structures, but as an optimisation for performing a large number of simultaneous updates on trees. The novelty of this part of our work is, thus, in providing a convenient abstraction to obtain efficient concurrent counterparts for arbitrary search structures that fit this profile.

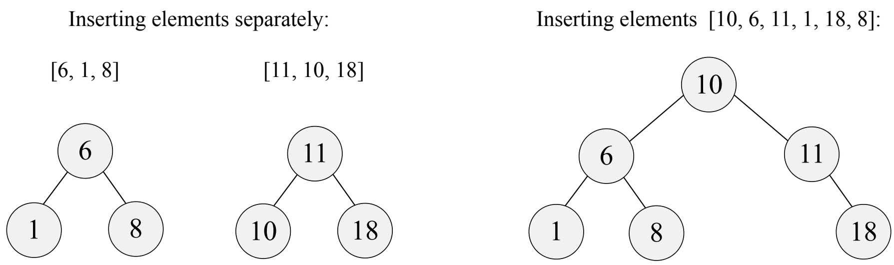


Fig. 11. Left: inserting elements into 2 empty separate trees. Right: inserting the same elements into a single empty tree (right). Joining the 2 trees on the left yields the single tree shown on the right, and splitting the tree on the right at element 11 yields the trees on the left.

The crux of the approach is in splitting a tree instance in a way that the combining the resulting sub-components after the parallel updates can be done with better than $O(n)$ complexity. In the case of binary search trees, a better complexity could be achieved if we had to join trees pair-wise where the maximum key element in the first tree, if any, is strictly less than the minimum key element in the second tree, so the range of keys of these trees do not overlap with each other. This would allow us not to explore those trees in full when joining, as hinted by the example in Fig. 11, where most of the subtree structure remains unchanged. To achieve that, we adopt the idea from the work of Blelloch et al. (2016), who were themselves elaborating on the work of Adams (1993) on implementing elegant yet efficient functional sets. They proved that `split` and `join` for balanced binary trees, such as red-black trees, AVL trees, and treaps, can have a worst-case time complexity of $O(\log n)$. In this scheme, joining two balanced binary trees consists of connecting them, comparing their balancing factor (*i.e.*, *height* for AVL trees, *black height* for red-black trees, and *priority* for treaps), and rebalancing them, if needed, by disconnecting root nodes from child nodes and applying rotations. Splitting applies a similar rebalancing strategy. Both these functions are called recursively only once per level, hence their logarithmic complexity.

4.1.1 A Functor for Split-Join Batching. To provide a convenient abstract interface for split/join batch parallelism, we define two modules signatures, **`Sequential`** (Fig. 12a) and **`Prebatch`** (Fig. 12b), that outline the required data types and functions to be provided by the user.

`Sequential` must contain the key type `kt` and the tree type `'a t`, as well as an implementation of the constructor `init` for creating a new data structure instance, `search`, `insert`, and `delete`. Note that operations that update the data structure should be done in-place, hence the final return type of ‘unit‘. We also do not require `'a t` to be the type of the “node”; it is often more convenient to make it a *wrapper over a root node* of a different type that recursively contains child nodes of that node type. The users are expected to implement **`Sequential`** by defining all basic sequential operations that can be used to interface with the data structure. Should there be no need for batching, one can simply invoke the functions here to use the data structure sequentially.

`Prebatch` is built on top of the **`Sequential`** module signature, and describes additional functions that the split-join functor will need to construct the batch-parallel version of the data structure:

- The function `compare` exposes the comparison function for the key type. For example, if we were using **`Int`** as the key type, we can simply define it like so: `let compare = Int.compare`.
- `set_root` swaps out the current root of the data structure. This is needed for updating the data structure in-place after joining its sub-components.
- `size_factor` returns the number that over-approximates the size of the current data structure. For instance, this can be the height of a tree. We will describe its use shortly.

```
module type Sequential = sig
  type kt
  type 'a t
  val init : unit -> 'a t
  val search : kt -> 'a t -> 'a option
  val insert : kt -> 'a -> 'a t -> unit
  val delete : kt -> 'a t -> unit
end
```

(a) The **Sequential** module signature

```
module type Prebatch = sig
  module S : Sequential
  val compare : S.kt -> S.kt -> int
  val set_root : 'a S.t -> 'a S.t -> unit
  val size_factor : 'a S.t -> int
  val split : 'a S.t -> S.kt -> 'a S.t * 'a S.t
  val join : 'a S.t -> 'a S.t -> 'a S.t
end
```

(b) The **Prebatch** module signature

Fig. 12. **Sequential** and **Prebatch** module signatures the for split-join functor.

- `split`: given a tree and a *pivot* value `k`, returns two trees with non-overlapping key ranges where the maximum key of the first tree is strictly less than `k`, and the minimum key of the second tree is equal or greater than `k`.
- `join`: provided two trees with non-overlapping key ranges and in ascending order based on those ranges, returns a single valid tree formed by joining the two input trees.

```
let run_batch t pool ops_array =
  let searches = ref [] in
  let inserts = ref [] in
  (* omitting deletions *)
  Array.iter (fun elt -> match elt with
  | Mk (Insert (key, vl), kont) ->
    (* safe to notify the client immediately *)
    kont (); inserts := (key,vl) :: !inserts
  | Mk (Search key, kont) ->
    searches := (key, kont) :: !searches
  ) ops_array;

  par_search pool t (Array.of_list !searches);
  par_insert pool t (Array.of_list !inserts)
```

Fig. 13. Split-join parallel batching

For each data structure that allows for efficient implementation of the split and join operations, the user need only implement these two modules and their functions. Once done, OBatcher's split-join functor can take over and define an explicitly-batched module, much like the one we have seen in Sec. 2.4. Let us examine its `run_batch` function defined in Fig. 13. For any given batch of operations, we start by separating different types of operations. We currently limit our implementation to search and insert operations, but it should be a fairly mechanical process to extend the functor to accommodate other effectful operations as well, following the handling of either batched searches or inserts as a template. After separation, we execute all searches, then all insertions. Even if we had interleaving search and insert requests in our initial batch, executing the operations this way is still correct from the perspective of linearisability, as we have discussed previously with the **Counter** example in Sec. 2.2. There is no particular pre-processing needed for batch parallel searches as those do not modify the data structure, and we simply use Domainslib's higher-order `parallel_for` function to dispatch the search operations in parallel.

Parallel execution of inserts is a bit more subtle; it relies on the various **Prebatch** functions defined earlier. We begin by checking whether (a) the present size of the data structure warrants splitting by invoking `size_factor` and whether (b) the number of insert operations is sufficiently high. We set some threshold for these two, with the intuition that smaller data structure sizes and smaller numbers of operations are not worth the extra overhead of splitting, parallelising, and rejoining, in which case we will simply perform the operations sequentially (we omit this part from our presentation). Otherwise, we perform the parallel insert procedure as shown Fig. 14. We select a number of random pivots based on the size of the batch and the sequential threshold (lines 2-7), and use them to split the tree (line 8) and partition the insert batch into ordered sub-batches

```
1  let par_insert ~pool s inserts =
2    let n =
3      Array.length inserts / seq_threshold + 1 in
4    (* assume that inserts are randomly ordered *)
5    let pivots =
6      Array.init n (fun i -> fst inserts.(i)) in
7    sort pivots;
8    let s_arr = split_multiple s pivots in
9    let sub_ranges =
10     partition pivots inserts in
11 parallel_for pool
12   ~start:0
13   ~finish:(Array.length sub_ranges)
14   ~body:(fun i ->
15     let range = sub_ranges.(i)
16     for j = fst range to snd range do
17       let k, v = inserts.(j) in
18       S.insert s_arr.(i) k v
19     done);
20 set_root t (join_multiple s_arr)
```

Fig. 14. Batch-parallel insertion in the split-join functor.

(lines 9-10). We then use Domainslib's `parallel_for` to perform each sub-batch of operations on its respective subtree in parallel (lines 16-19). Finally, we rejoin the split trees together (line 20). Our implementation takes advantage of some other opportunities for parallelisation, namely, sorting the random pivots via a classic implementation of parallel merge sort, and partitioning the insertion operations with a parallel partition procedure inspired by the one used in the QuickSort algorithm.

4.1.2 Case Study: Red-Black Tree. As a concrete example of the split-join strategy for batch parallelism and its respective functor in action, let us take a look at the red-black (RB) tree (Bayer 1972), a classic self-balancing binary search tree data structure. In addition to searching and insertion of elements, its typical implementations support efficient deletion, minimum, and maximum—all of them enjoying logarithmic worst-case time complexity.

Sequential red-black tree overview. A red-black tree is an *approximately* balanced binary search tree, meaning that it is not perfectly balanced, but instead guarantees that no root-to-leaf path *is more than twice* as long as any other root-to-leaf path. This is achieved by assigning each node a colour, which can be either red or black, and ensuring that the following invariants are upheld: (1) every leaf (not containing any key) is black, (2) if a node is red, both its immediate children must be black, and (3) each path from a given node to any leaf must have the same black height, *i.e.*, the same number of black nodes. Some presentations also add an extra condition: the root node must be black. We will omit this rule for our implementation, as it is not essential for the desired time complexity, and we must allow a red root for the split and join functions later on. Aside from this omission, our implementation follows the red-black tree algorithms as described in the standard textbook by Cormen et al. (2009). Fig. 15 shows the balanced binary tree that we've previously seen in Fig. 11, but as a red-black tree, with some of its nodes are coloured in red, so that the tree invariants stated above are respected.

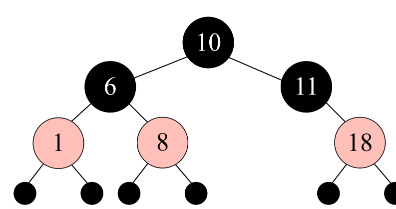


Fig. 15. Example of a red-black tree

Following the **`Sequential`** signature from Fig. 12a, whose effectful functions (*e.g.*, `insert`) modify the data structure in-place, we implement a wrapper type `'a t` around the root node of the red-black tree. Searching a red-black tree follows the same procedure as for an unbalanced binary search tree. Inserting a node involves recursively traversing the tree like the search operation and adding a new node at the leaf level, after which, the tree might be repaired by recolouring and rotating as needed (we refer the reader to Chapter 13 of the textbook by Cormen et al. (2009) for the details).

Batch-parallel red-black tree overview. The logic of a batch-parallel red-black tree follows the general split-join strategy from Sec. 4.1.1, requiring one to implement the `set_root`, `size_factor`, `split`, and `join` functions to obtain a fully functional batch-parallel version of the data structure.

The `set_root` function is straightforward: we simply replace the root node in the red-black tree wrapper type. For the sake of other functions, we augment our sequential implementation with an additional piece of information stored in each node: the *black height*, *i.e.*, the number of black nodes on the paths from that node to each leaf (Blelloch et al. 2016). This will serve as the basis for the `size_factor` function: we can use the black height as an estimate of the size of the tree. This addition also enables implementations of `split` and `join` operations. For these functions, we faithfully recreate the algorithms as described in pseudocode by Blelloch et al., where rebalancing and recolouring occur depending on the black height difference between the input trees. Storing black heights avoids the additional $O(\log n)$ cost of counting the number of black nodes every time we need this information, hence keeping `split` and `join` to $O(\log n)$ time complexity as well.

4.1.3 Other Split-Join Data Structures. As shown, balanced binary search trees are especially well-suited for the split-join batch-parallel paradigm, subject to their own `split` and `join` functions. The work by Blelloch et al. (2016) also provides blueprints of these functions for other types of search structures: AVL trees (Adelson-Velsky and Landis 1962) and treaps (Seidel and Aragon 1996).

Like an RB tree, an AVL tree balances itself through rotations. It does not have any colour code, and instead preserves the invariant that at any given node in the tree, its left and right sub-trees have a height difference of at most one. A *treap*, meanwhile, is a probabilistically balanced tree where each node is randomly assigned a priority number, and we rotate the tree after each update to ensure that the priority number of each node is higher than the prioriy numbers of its child nodes, essentially, recreating a max heap based on the random priority number.

We implemented their batch-parallel versions just like with the RB tree by defining their `split` and `join` functions as described by Blelloch et al. (2016). To implement the `size_factor` function, we simply use the tree height already stored in AVL tree nodes, and augment the treap nodes with stored tree heights as well. Our accompanying code provides the respective working implementations.

4.2 Expose-Repair Batching Strategy

The split-join strategy discussed in Sec. 4.1 is beneficial for the search structures that admit sublinear-time changes in their shape without breaking their invariants, such as rebalancing. For instance, we rely on being able to move nodes around in binary search trees to split and rejoin them. Not every search structure lends itself well to being batch-parallelised this way, meaning that they cannot be split into sub-structures of the same shape, or that joining them would induce a prohibitively large overhead undoing the performace improvement gained from parallelism.

This is the case for search structures, such as bitwise tries, that use string/binary prefixes or hashes to store key values: in those structures the *position* of an element in the data structure *is* its key. Therefore, "physically" splitting such a structure tree will inevitably require one to deal with complex re-indexing logic. On the flipside, we can argue that such a search structure is even better suited to a batch-parallel implementation since each update should only affect its predictable, *localised* area that is the same regardless of what other keys populate the data structure.

Making such "position-based" search structures batch-parallel might not be as straightforward as it seems. Advanced examples like the van Emde Boas tree (1975), the x-fast trie, and the y-fast trie (Willard 1983), which we will showcase for this section, contain additional metadata, pointers, or even entire secondary sub-structures to achieve the promised sub-logarithmic time complexity of their operations in a sequential case. Those sub-structures must be accounted for, restored, and/or updated both before and after running a batch.

```
module type Sequential = sig
  type kt
  type t
  val init : int -> t
  val mem : t -> kt -> bool
  val insert : t -> kt -> unit
  val delete : t -> kt -> unit
  val predecessor : t -> kt -> kt option
  val successor : t -> kt -> kt option
end
```

(a) The **Sequential** module signature

```
module type Prebatch = sig
  module S : Sequential
  type dt
  val compare : S.kt -> S.kt -> int
  val expose :
    S.t -> S.kt array -> S.kt array * dt
  val repair : S.t -> dt -> unit
  val insert_range :
    S.t -> S.kt array -> dt -> int * int -> unit
 end
```

(b) The **Prebatch** module signature (simplified)

Fig. 16. **Sequential** and **Prebatch** module signatures for the expose-repair functor.

In this section, we present a novel strategy for batch parallelism, dubbed *expose-repair*, which is aimed to address such search structures. It centres around first "exposing", or preparing the structure before each batch of operations, so as to make sure parallel operations in each its localised area do not affect each other. Then, we "repair" the result after processing the batch of operations, to re-establish the metadata or sub-structures covering the entire data structure.

4.2.1 Expose-Repair Functor Overview. Similarly to the split-join functor from Sec. 4.1, for expose-repair we define two modules signatures, **Sequential** and **Prebatch** that need to be instantiated by the user. For simplicity, we phrase the **Sequential** interface as a *set* rather than as a key/value, *i.e.*, it only stores the keys of the type `kt`, following the presentation from the standard textbook (Cormen et al. 2009, Chapter 20). The key/value map-like functionality can be easily restored by associating the "satellite data" with the keys. As the result of this design, the **Sequential** signature shown in Fig. 16a offers the `mem` function instead of `search`. The signature also features the `predecessor` and `successor` functions, as one of the main selling points of the fast search structures, such as van Emde Boas tree, the x-fast and the y-fast tries, which all offer the $O(\log\log u)$ time complexity for these operations (where u is the largest integer key that can be stored in the structure). Just like in the case of split-join strategy, the pure query operations like `mem`, `predecessor`, and `successor` can be dispatched in parallel in a separate stage of the batch processing, without any special preparation.

The signature for the **Prebatch** module is best explained in parallel with the implementation of a parallel executor for the effectful operations in the respective expose-repair functor provided by OBatcher, which makes use of the **Prebatch** components. As a characteristic example, consider the implementation of parallel insertion via expose-repair strategy shown in Fig. 17. It starts by checking if the number of insert operations in the batch exceeds the sequential threshold, and if it is below, performs the operations sequentially (we omit this part from the listing for brevity); otherwise, its continues with the the batch processing. To do so, it first obtains `n` sorted random "tentative pivots" from the batch of operations, and transforms them into pivots we can use to partition our batch of operations (lines 2-7). Unlike the split-join functor, which takes random elements from the batch of operations as pivots on which to split the trees (Sec. 4.1.1), in the case of position-based structures we would want pivots that can separate the data structure into logical parts consistent with their inner layout and the recursive workings of their respective sequential operations. For instance, in the case of the van Emde Boas tree, those pivots would be *multiples of the square root of the size of its universe* (*i.e.*, the set of all its possible keys), so that we can divide up a cluster of trees without dissecting any tree in the middle (more details are given in Sec. 4.2.2). Next, the implementation prepares the data structure for batch processing

```
1  let par_insert ~pool t inserts =
2    (* omitted: checking if the batch size is larger than seq_threshold *)
3    let n = Array.length inserts / seq_threshold + 1 in
4    (* assume that inserts are randomly ordered *)
5    let pivot_seeds = Array.init n (fun i -> fst inserts.(i)) in
6    sort pivot_seeds;
7    let pivots, dt = expose t pivot_seeds in
8    let sub_batch_ranges = partition pivots inserts in
9    parallel_for pool ~start:0 ~finish:(Array.length sub_batch_ranges)
10     ~body:(fun i -> insert_range t inserts dt sub_batch_ranges.(i));
11   repair t dt
```

Fig. 17. Batch-parallel insertion in the expose-repair functor.

where needed and partitions the batch of operations using the obtained pivots (lines 7-8). The preparation can take the form of (but is not limited to) pre-initialising parts of the data structure, or temporarily removing some pointers between nodes belonging of different sections of the data structure. This process might both rely on and/or inform the creation of the pivots. The partitioning (line 8) only depends on the values of the pivots and the arguments of the insert operations, hence is not structure-specific. Next, the sub-batches of operations resulting from the partitioning are run in parallel on the same pre-processed data structure (lines 9-10). It is done in the assumption that `expose` has indeed returned the pivots in a way that split the range of the insertions in a way so that operations in different sub-batches would not interfere with each other. Finally, the initial data structure `t` is repaired by, *e.g.*, removing unused pre-initialised parts, updating metadata, and restoring auxiliary pointers between its sub-parts (line 11).

The code in Fig. 17 relies on the following components of the **`Prebatch`** functor from Fig. 16b:

- An abstract type `dt`, which is specific for each particular search structure and is used to store supplementary information for batching, if needed (*cf.* Sec. 4.2.2 for a concrete example).
- The function `expose` takes as its input the data structure itself and an array of "tentative pivots". Having those, it (a) prepares the data structure in-place for batch processing and (b) returns an array of the pivot values used to partition our batch of operations. We combine these two procedures, as they can be very much intertwined with one another.
- The function `repair` repairs the data structure after all updates have finished.
- Finally, `insert_range` performs a sequence of insertions into the exposed data structure sequentially. It takes as input the data structure, a reference to the whole batch of insertions with the range of the sub-batch (to avoid reallocating a new array for each sub-batch), and extra information in the form of `dt`. Note that `insert_range` takes the whole data structure as input, since inserting a specific range of values should keep the effects localised and therefore should not affect any other instance of `insert_range` running in parallel.

Conceptually, the correctness of an expose-repair strategy is simpler than split-join for trees. Broadly speaking, one can argue for the correctness of an implementation from the following assumptions about the operations: 1) first, `expose` should preserve the elements in the structure and should only logically split the structure into disjoint "chunks" via pivots, 2) second, for a given range, the `insert_range` function should not affect "chunks" outside of it, and finally 3) `repair` should not lose elements and restores any data structure invariants. We illustrate this in more detail in our argument for the correctness of the van Emde Boas Tree in the next section. Supporting parallel deletions using the expose-repair strategy would require implementation of a function with a signature similar to `insert_range`, which we have omitted for the sake of brevity.

4.2.2 Case Study: Van Emde Boas Tree. The van Emde Boas tree (vEB tree for short) represents a priority queue and was conceived as a way to resolve bottlenecks in in query operations (*e.g.*, membership, predecessor, successor) for ordered, random access sets (van Emde Boas 1975). When implemented as binary trees, such as RB tree, these operations have $O(\log n)$ time complexity. The vEB tree instead allows for membership, predecessor, and successor queries to be done in $O(\log\log u)$ time, where u is the universe size, or the number of all possible key values the vEB tree might store. This logarithmic speedup over balanced binary trees is achieved at the cost of space efficiency, with the vEB tree occupying $O(u)$ space regardless of how many elements its contains.

Sequential van Emde Boas tree overview. A vEB tree is structured recursively, with each node containing (1) the universe size u at that node, (2) the minimum value of the tree at that node, (3) the maximum value of the tree at that node, (4) an array `cluster` of $\sqrt{u}$ vEB tree nodes, each of which contains a tree of universe size $\sqrt{u}$ so that a vEB tree of an index i would contain keys in the range $[i\sqrt{u}, (i+1)\sqrt{u}-1]$, and (5) a summary vEB tree of size $\sqrt{u}$ storing the indices of vEB trees in the cluster array that are non-empty, *i.e.*, have at least one element.

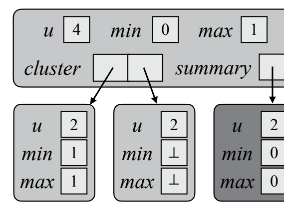


Fig. 18. A vEB tree containing $\{0, 1\}$.

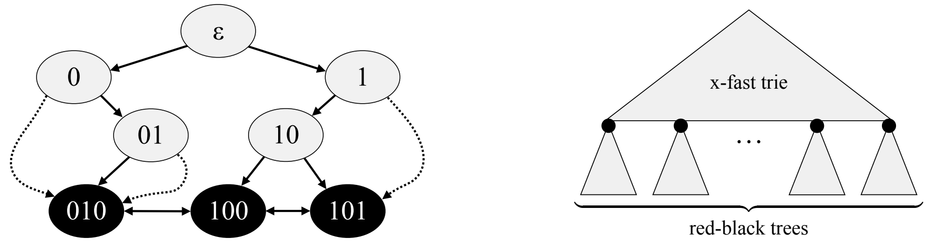
Consider the example vEB tree in Fig. 18. It has the universe size of 4, so its possible key values range from 0 to 3. In this case, it only contains the keys 0 and 1: 0 is stored in the *minimum* field of the root node and is thus is not present in the vEB nodes below (the minimum field of a vEB node is not just metadata, but always contains the key itself). The *maximum* field of the root vEB node shows 1, but this is *not* the key value itself, but is just metadata. Looking down, the left child node, corresponding to the index 0 of the root cluster, itself has no cluster, and both its minimum and the maximum fields store 1. It is a *base case* vEB node, whose universe size is 2, and whose maximum field *can* contain the key itself. The vEB node at the root cluster index 1 is empty, as shown with both its minimum and maximum being $\perp$. Finally, the root features an additional *summary* vEB node, which just contains 0, meaning that only the vEB child node at the position 0 of the root cluster is non-empty. Though it may seem odd in this example to "summarise" a cluster of size 2, for larger clusters it allows one to find the first non-empty cluster in $O(\log\log u)$ time, enabling fast predecessor and successor queries. We omit detailed descriptions of sequential vEB operations, and refer the reader to Chapter 20 of Cormen et al. (2009) or to our accompanying implementation.

Batch-parallel van Emde Boas tree overview. The vEB tree lends itself naturally to batch processing. We note that since there are $\sqrt{u}$ trees available in the cluster at the root level, for any reasonable choice of universe size, there will be a sufficient number of vEB trees even at the first level, that we need not look further below for parallelising our batched operations on the vEB tree.

The code of the `expose` function for vEB tree is shown in Fig. 19. We will not need any supplementary information, so the type `dt` is just **`unit`**. We do however need to identify how to create appropriate pivots from random keys in our batch of insert/delete operations in order to split the vEB tree into logical chunks. For this, we simply transform each pivot key into the minimum possible key for its target sub-vEB tree in the root-level cluster array, which is done using a standard helper function `high` taken from Cormen et al. (2009) (Chapter 20). This ensures that keys in different sub-batches are never inserted in the same vEB tree, nor in the same cluster sub-array. At the end, we deduplicate the final list of pivots, as

```
let expose t arr =
  let size_cluster =
    lower_sqrt t.uni_size in
  let pivots =
    Array.init
      (Array.length arr)
      (fun i ->
        high t arr.(i) * size_cluster)
  (dedup pivots, ())
```

Fig. 19. The vEB tree expose function



(a) The x-fast trie (b) The y-fast trie

Fig. 20. X-fast trie and Y-fast tries. For the x-fast trie in Fig. 20a, solid double arrows indicate pointers from each leaf to the previous and the next leaf, while dashed single arrows indicate descendant pointers.

two different keys going to the same vEB tree will yield the same pivot. For each key `k` to be inserted in the sub-batch of operations defined by the input range, `insert_range` invokes the sequential insertion procedure, but *does not* update the summary node, the minimum, or the maximum at the root level. This last update is exactly what is done by the respective `repair` function.

With respect to the correctness of this implementation, we find that it follows the conditions stated at the end of Sec. 4.2.1 and thus yields a correct vEB at the end of batched operations:

(1) The implementation of `expose` for the vEB tree does not modify the data structure at all, but merely returns the ranges for non-overlapping cluster sub-arrays.
(2) Since we made pivots such that keys in different sub-batches cannot be inserted in the same vEB tree or even the same cluster sub-array, each `insert_range` call cannot affect anything other than its own "chunk" of the vEB tree.
(3) The `repair` function restores metadata at the root node by updating the minimum, maximum, and summary tree. This is the only we need to update at the end, as each cluster node vEB tree is already valid at the end of parallel insertion.

4.2.3 Other Expose-Repair Data Structures. In addition to vEB, we have instantiated the expose-repair for two more position-based search structures: the x-fast and y-fast trie by Willard (1983). Both these structures were designed to preserve the query time complexity of the vEB while only using $O(n \log u)$ and $O(n)$ space respectively, where n is the number of elements in the trie.

In an x-fast trie (Fig. 20a), all full unsigned integer keys are stored at the same leaf level, with there being as many intermediate layers as there are bits representing these integers. Each leaf node points to its successor and/or predecessor leaf node. Each internal node with no left child contains a pointer to the smallest leaf in its right subtree, similarly, each internals node with no right child contains a pointer pointer to the largest leaf in its left subtre; in both cases those are referred to as *descendant* pointers. Typically, at each layer an x-trie has a hash table to accelerate queries. As a proof of concept, our implementation uses arrays, which worsens the worst-case x-fast trie's space complexity, but allows for a relatively simple expose-repair implementation: we `expose` the sub-trie by removing pointers and adding intermediate nodes up until the layer determined by the number of pivots, and partition our operations among the nodes at that layer.

To turn the x-fast trie into a y-fast trie (Fig. 20b), we replace the simple leaf nodes with a forest of red-black trees. The size of each RB tree has an expected size of $\log u$, and it is split as needed to maintain that size. Implementing a batch-parallel version of the y-fast trie using the expose-repair functor is not very complicated. For insertions, `expose` determines the ranges for the keys to be inserted and RB trees that each parallel task should cover based on random pivots from the batch of operations, dispatching those operations sequentially within `insert_range`. At the end, `repair` checks the size of each resulting RB tree, splitting them where needed to respect the size bound.

4.3 Case Studies with Ad-Hoc Batch Parallelism

We conclude this section by presenting two implementations of ad-hoc batch parallelism that do not immediately follow either of the split-join or expose-repair strategies: a B-tree and a stateful Datalog solver. We have additionally implemented a batch-parallel version of a skip list (Pugh 1990) by adopting the design proposed by Agrawal et al. (2014), but omit its description for the sake of space; a curious reader may find it in the extended version of this paper (Le et al. 2024a).

4.3.1 B-Tree. A B-tree is a widely-used tree data structure that can be used to implement an efficient set or map interface (Bayer and McCreight 1972). The B-tree's efficiency arises from the fact that its operations are carefully structured to always maintain the balance of the tree and try to preserve a good distribution of values across the tree, thereby ensuring a logarithmic lookup time. Fig. 21 depicts the general structure of a B-tree. The nodes in a B-tree contain 2 components: a sorted sequence of keys $k_1, \ldots, k_n$ stored in the node, and a sequence of sub-trees $c_1, \ldots c_{n+1}$ corresponding to the node's children; leaf nodes contain keys, but no children. The B-tree forms a search tree in the fact that the keys in a node define *intervals* that bound the contents of children: a child c_i will only contain keys that lie in the range $k_{i-1} \leq v < k_i$. Finally, to provide a consistent performance, the B-tree enforces the constraint that for any node, all of its sub-trees must all have the same height (*i.e.*, the tree is balanced), and that every interior node will have a number of keys between $t - 1$ and $2t - 1$, where t, the branching factor, is a global parameter of the tree, ensuring that values are evenly distributed.

Adding new values to a B-tree requires care to preserve the its invariants. The high-level intuition for an insert is to recursively traverse down the tree to find the leaf into which to insert the value—but what if the leaf is already at full capacity (*i.e.*, it contains $2t-1$ keys)? To solve this, insertion relies on the *splitting* operation: when a particular node in the data structure has reached full capacity, we can gain additional capacity while preserving the structure's invariants by splitting the node into two and adding a new key and child to the node's parent. While splitting requires modifying the parent, we can prevent this from bubbling up the entire B-tree by maintaining an invariant that the node being visited during the recursive traversal has *at least one free space*, thus, preemptively splitting a sub-tree before descending down the tree. In this way, even if inserting a value into a node requires splitting, we can be guaranteed that *no further splits* higher up in the tree would be required to restore the B-tree's intrinsic invariants.

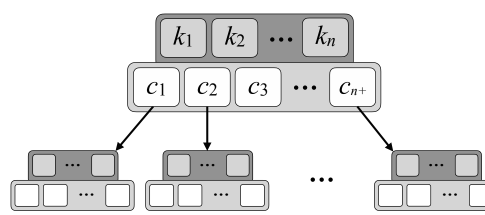


Fig. 21. Structure of a B-tree

A batch-parallel B-tree. Implementing a batch-parallel search in B-tree is straightforward: when processing a batch of searches, first dispatch the values that occur at the current node, partition the remaining values by the children, and recurse on each sub-tree in parallel. At a high level, the same divide-and-conquer strategy should be applicable to B-tree inserts: when inserting a batch of values into a B-tree, we can partition the values by sub-trees and recurse on each child in parallel. The problem with this strategy is in ensuring that the nodes of the B-tree never become over-full during the inserts, as this causes them to split, possibly causing a cascade of splits to "bubble up", making them interfere with other parallel operations.

Inserting n elements into a given node can require splitting up to n children in the worst case, *i.e.*, when each of the elements falls into a *different* child, each of which itself needs splitting. If we only look at the contents of a single node, then the maximum capacity of any node will be at most t: this follows from the fact that the B-tree requires that every node has a number of keys between

$t-1$ and $2t-1$, so starting from the minimum and encountering t splits would already cause the node hit the maximum number of keys. As the size of an input batch is typically far larger than t, a constant parameter of the tree, this overly-conservative bound would hamper the parallelism.

The key insight in our design is in defining a novel notion of capacity for a B-tree node (*cf.* Fig. 22) that is more precise but still relatively cheap to compute. For a leaf node, the capacity is simply the number of elements that can be inserted before it hits max capacity. For an interior node, notice that each time one of its children splits, it increases the number of keys that the node has by 1, until it reaches max-capacity at which point an additional element could cause the node to require splitting. As such, we can conservatively bound the capacity of a node by the capacity of its children plus 1, that is, the minimum number of elements to cause a child to split, multiplied by the free slots it has, the number of times it can allow its children to split. By using the minimum here, we lose some precision in our bound, but this also means that the value of this statistic can be computed without imposing significant overhead.

$$
\begin{aligned}
\mathrm{nkeys}(n) &= \text{number of keys in the node } n \\
\mathrm{child}(n,i) &= i^{\text{th}} \text{ child of the node } n \\
\mathrm{free}(n) &= 2t-1-\mathrm{nkeys}(n) \\
\mathrm{cap}(n) &= \begin{cases} \mathrm{free}(n) & n \text{ is a leaf} \\ (1+\min_i \mathrm{cap}(\mathrm{child}(n,i))) \times \mathrm{free}(n) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 22. Capacity of a B-tree

Using this new definition of capacity, we can construct a batch-parallel B-tree insert using the divide-and-conquer strategy. In particular, after partitioning the batches by the children, if the size of a batch exceeds the capacity for the sub-tree, then we preemptively split the sub-tree until the desired capacity is reached. Once all the splits have been performed, we can then recursively insert into the children in parallel, returning to the standard divide and conquer strategy.

Putting it all together, we incorporate both these search and insert implementations to instantiate the explicitly batched interface of OBatcher and construct an efficient batch-parallel B-tree (right). We first partition the input batch, splitting out the searches and inserts to be handled separately. These partitions are then pre-sorted using an off-the-shelf parallel merge-sort implementation. Finally, we execute the requested batch by first dispatching the searches using our batch-parallel search implementation, and then subsequently dispatching the inserts using the corresponding batch-parallel insert one.

```
let run_batch t pool ops =
  let searches, inserts =
     (* partition batch *) in
  sort pool searches;
  sort pool inserts;
  par_search t pool searches;
  par_insert t pool inserts
```

4.3.2 Datalog Solver. As our final case study, we demonstrate the generality of OBatcher by investigating its use for constructing a *thread-safe* and *efficient* wrapper over a popular OCaml Datalog library (Cruanes 2024) with minimal additional implementation effort. The main interface to the library is through a type `db` encoding Datalog databases is presented on the right. Here, each database `db` represents a collection of facts known to be true, where each fact is a relation applied to zero or more symbols, *e.g.*, `edge("a","b")` might encode knowledge that an edge exists between nodes `"a"` and `"b"` in some graph. Users can extend a database using the `insert` operation, submitting new facts or adding deductive rules that describe how to deduce new facts from existing ones. Once a database has been constructed, `datalog` allows users to query the set of facts in the database. By submitting terms `query`, *i.e.*, `edge("a","b")` or `connected("x",y)`, the database will return a list of all facts that match the query, answering questions such as *is there an* `edge` *between* `"a"` *and* `"b"`*?* or *what nodes are connected to* `"x"`*?* The `datalog` library was not written with a multicore environment in mind. It makes pervasive use of

```
module Datalog : sig
  type db
  type term
  val insert: db -> term -> unit
  val query: db -> term -> term list
end
```

mutation and shared state that would be challenging to port to a fine-grained implementation. In contrast, implementing a batched interface to the library is comparatively simple: when handling a batch of requests, we first dispatch all queries in parallel, and then evaluate each of the inserts sequentially. We use this simple idea to instantiate OBATCHER's interface with the `datalog` library.

5 Evaluation

The main intended use for concurrent search structures is to allow concurrent tasks to exchange data without introducing communication bottlenecks. Since such structures form the majority of our case studies, in Sec. 5.1 we provide an extensive evaluation of their throughput trends. In Sec. 5.2, we evaluate the performance of the concurrent Datalog solver. The goal of our experiments is not to claim the maximum performance, which still requires carefully crafted concurrency; instead, we aim to show that our approach provides reasonable performance with less work.

All the reported benchmark results were obtained by running the experiments on an AWS EC2 `c7i.12xlarge` server instance equipped x86 Intel® Xeon® Scalable (Sapphire Rapids) processor with 24 physical cores and 96 GB of memory, running Ubuntu 22.02 with OCaml 5.1.1.

5.1 Benchmarking the Concurrent Search Structures

We evaluate the throughput for all eight search structures from Sec. 4 (RB tree, AVL tree, treap, van Emde Boas tree, x-fast and y-fast tries, B-tree, and skip list) on the same set of benchmarks. For each benchmark, we fix the number of initial elements in the data structure at 2,000,000 and the workload size to be 1,000,000 operations. We experiment with four different workload setups: inserts only, searches only, 50%/50% and 90%/10% search/insert split. Each operation of the workload is submitted to Domainslib's thread pool as a separate concurrent task. Each data point takes the average of five runs, performed after five warm-up runs. We compare the performance of our batch parallel implementations with their respective coarse-grained and sequential implementations, varying the number of domains – thin wrappers over system-level threads provided by the OCaml runtime. Fig. 23 shows our results, allowing us to draw several observations detailed below.

General performance trends. First, we observe that in nearly every benchmark, our batch-parallel generated search structures outperform their coarse-grained counterparts by a significant margin starting from two domains, with the gap widening as we increase the number of domains. The gap is particularly evident for insertion-heavy benchmarks, where we see the batched implementations match or even outperform the sequential ones when more domains are available. Fully sequential executions typically outperform the batch-parallel ones for searches, even for higher domains and by a significant margin. This is likely due to the efficiency of searches (even when done sequentially), combined with the comparatively large overheads for creating/managing parallel tasks.

Second, we observe that in nearly all cases, in the case of a single domain, a batch-parallel implementation shows a lower throughput than the coarse-grained one. This is because with a single domain, the batch-processing routine is launched for every submitted operation after the waiting period between batches in the `try_launch` function (Fig. 8), due to not meeting the minimum batch size—as there are no other threads contributing operations. Therefore, we conclude that, without further optimisation, batch-parallelism only pays off in strictly multi-threaded scenarios.

Third, we observe that our x-fast and y-fast tries are in general less performant than our binary trees (AVL and RB), despite their superior asymptotic time complexity, even when considering only the sequential implementations. We conjecture that this is due to their much heavier memory use than that of the binary trees, since each key requires *multiple nodes* to represent, and so adding nodes would take more time putting additional stress on OCaml's concurrent memory allocator.

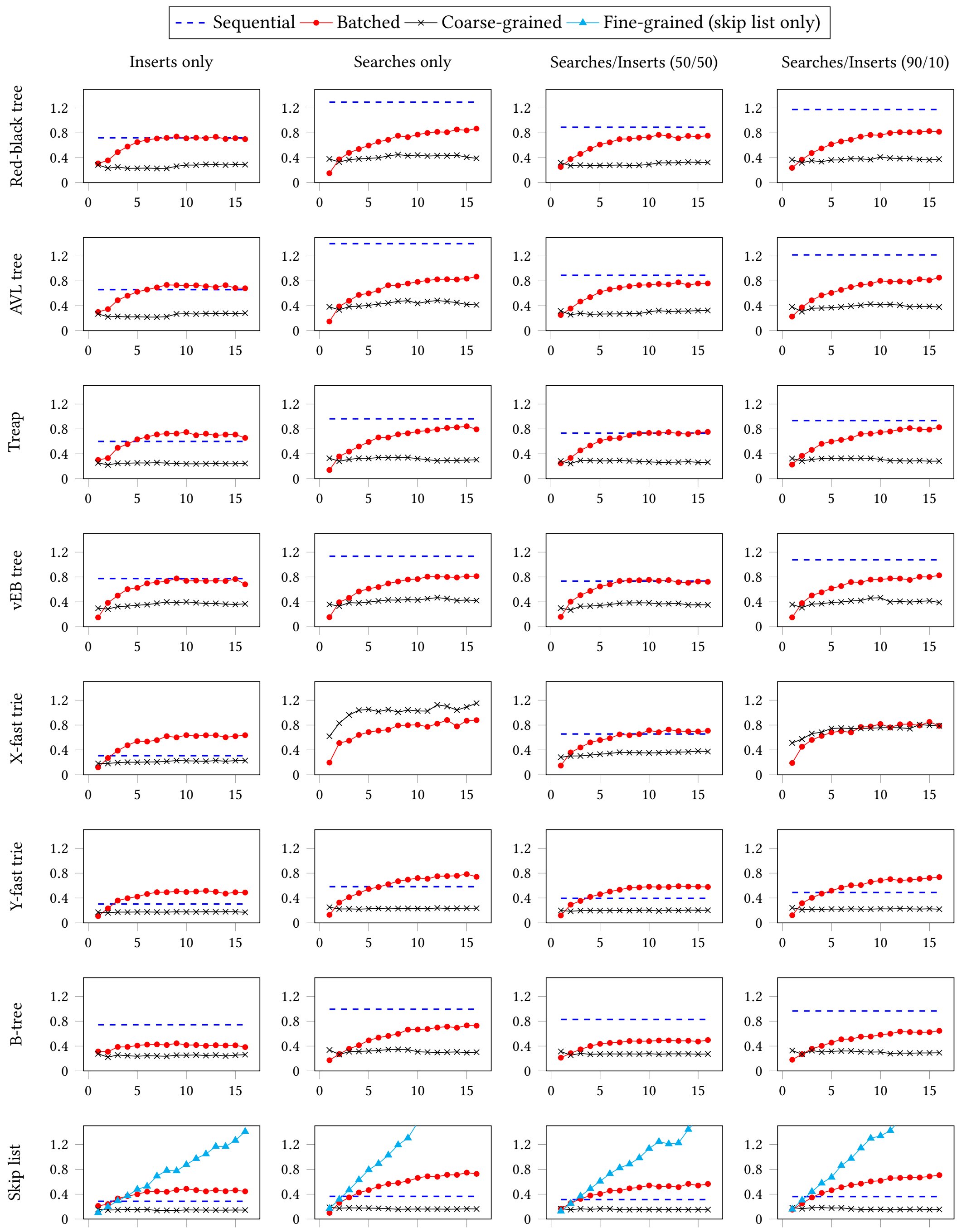


Fig. 23. Throughput comparison for concurrent search structures from Sec. 4. The x-axes represent the numbers of involved domains/cores, and y-axes show throughput in millions of operations per second.

Finally, we note that almost all the batch-parallel data structures show reasonable speed-ups: the throughput increases with the number of domains, up through approximately 8 domains. From that point onwards, throughput grows more slowly with the number of domains. This slowing growth at larger numbers of domains presumably stems from the increased synchronisation overhead.

Concurrent searching in an x-fast trie. The x-fast trie search-only and 90/10 search-insert split benchmarks are the only scenarios in our suite, where our batched implementation struggles to even match the coarse-grained implementation. This is the only data structure where search time is $O(1)$. Indeed, the sequential benchmark throughput for searching in an x-fast trie is orders of magnitude higher than of the other data structures, exhibiting around $22 \cdot 10^6$ op/sec in the search-only benchmark, and around $2.8 \cdot 10^6$ op/sec for the 90/10 search-insert split benchmark. We can surmise that this is a rare case where the underlying operation is so fast that, for a small number of threads, the lock contention imposes a smaller overall overhead than starting a batch.

Comparison with a fine-grained skip list. We have also compared our batched skip list implementation to the fine-grained lazy skip list by Herlihy et al. (2007), which we have ported from Java to OCaml. Sadly, Fig. 23 clearly indicates the growing performance gap between the fine-grained and the batch-parallel implementation. However, instead of simply admitting the crushing defeat by the giants of the multiprocessor programming, we are going to make a few observations that should encourage the reader to not dismiss batching as a methodology for implementing concurrent structures. 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 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 batch-parallel implementation does not feature any synchronisation primitives whatsoever, relying exclusively on `parallel_for`. It should also be noted that the batch-parallel implementation can also benefit from various sequential optimizations. In the context of skip lists, preprocessing steps, such as sorting inserts or removing k-smallest elements are analytically more efficient than their iterative counterparts (Hendler et al. 2010). More generally, deduplication of requests could also stand to improve throughput. Our current implementation only uses pre-sorting as an optimisation but could be improved further.

5.2 Benchmarking the Concurrent Datalog Solver

For our experiment with the `datalog` library, we implement a typical graph-connectedness constraint system on top of the Datalog database, using a graph of 200 nodes, with 30,000 initial edges, and submit requests that either add new edges or query about the connectedness of two nodes in the graph. All implementations are evaluated on a workload of 50,000 concurrent requests, using a standard 90%/10% split between queries and inserts, varying the number of domains/cores and taking the average of 10 runs for each configuration.

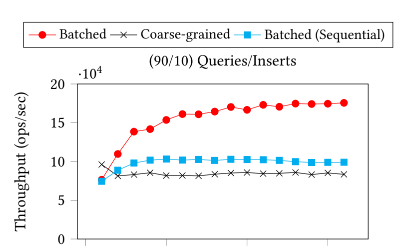


Fig. 24. Datalog solver throughput.

Fig. 24 presents the results of this evaluation. As can be seen, our construction is efficient and scales gracefully as the number of domains increases, while the corresponding coarse-grained wrapper's performance gradually degrades as more domains are added, suffering from increasing lock contention. Our plot also includes the throughput of a naïve batch-parallel implementation of a `datalog` wrapper, which simply executes all operations, queries and inserts alike, in a sequence.

Interestingly, its throughput still outperforms the coarse-grained version, showing the benefits of the amortised cost of locking provided by the batched interface yet it is unable to exploit parallelism.

6 Related Work

Flat combining and delegation-based locking. The technique of flat combining itself draws from long-running line of research into delegation-based locking. These works all investigate minimising lock contention by delegating concurrent requests to be executed under a dedicated worker.

Oyama et al. (2000) were the first to introduce this strategy, and considered using a simple LIFO queue to keep track of additional requests that may have accumulated while a client was in a critical section. Hendler et al. (2010) developed the flat combiner by extending this design and pre-allocating empty entries in the task queue for each thread thereby minimising contention between clients when appending tasks. To avoid situations where the queue is mostly empty, the flat combiner design also incorporates strategies for periodically compacting the queue and only keeping entries from threads that have recently submitted tasks. Subsequent works have built on this design and investigated optimising the configurations for factors such as cache locality (Fatourou and Kallimanis 2011; Roghanchi et al. 2017), and specialised multicore hardware (Lozi et al. 2016). Most recently, Gupta et al. (2023) developed a technique to transparently incorporate delegation-based locking into existing systems by constructing a dynamic lock that manipulates the call stack at runtime to suspend the client and send arbitrary critical section code to be executed by a dedicated worker, allowing delegation to be introduced into a system without significant source-level changes.

Delegation-based locking research has primarily focused on mitigating lock contention and, as such, prior work has not considered exploiting parallelism within the worker itself as in OBatcher. The insight in this work is that these techniques from delegation-based locking can be combined with `async`/`await` primitives to support implicit-batching in a lightweight fashion.

Batch parallelism and data parallelism. Data parallelism has long been a common approach for implementing parallel computations, and often it is implemented in a "batched" manner. For example, if you have a function that needs to be applied to a large number of data records, it is quite natural to divide the records into batches to apply the function in parallel.

Batch-parallel data structures do indeed bear a resemblance to data-parallel collections available as libraries, *e.g.*, in Haskell (Peyton Jones et al. 2008) and Scala (Prokopec et al. 2011). The main difference between data parallelism and batch parallelism is, perhaps, subtle. The former approach focuses on optimising the execution of *one* aggregate-style bulk operation (*e.g.*, `map`, `reduce`, `filter`, *etc.*) at a time by exploiting possibilities for parallelism allowed by the structure itself. From the perspective of the user a call to a data-parallel `map` is the same as a call to a sequential `map`, which just works faster. Importantly, neither of those calls assume concurrent interference from other calls made to the same collection. Finally, for data parallelism, it is typically easy for the programmer to construct explicit batches, as the parallelism to exploit has a simple structure.

By contrast, batch parallelism solves the problem of running *multiple* concurrent operations to a data structure in the most efficient way, while being aware of possible interference between them. Batch parallelism operates on structured data, extracting parallelism from whatever operations happen to arrive, on the fly. Moreover, the benefit of implicit batch parallelism is that it can handle situations where the batch structure is not, a priori, clear or easy to determine. That said, for each particular such operation in its batch, a batch-parallel implementation can make use of data-parallel techniques, even though we did not exercise this possibility in the scope of this work.

Other batch-parallel data structures. Over the last several years, researchers have developed a variety of batch-parallel data structures that focus on the problem of how to process a batch of data structure updates in parallel without being concerned how the batches are being constructed in the first place. The existence of these types of batch-parallel data structures provides good motivation for the creation of a generic implicit batching library like the one provided by OBatcher.

As of today, batch-parallel structures and algorithms are an active research area. Tseng et al. (2019) developed dynamic trees that could support batch update operations, and then used them to maintain Euler tours. Acar et al. (2020) also explored the problem of maintaining dynamic trees subject to batches of updates, handling a broader class of queries. Tseng et al. (2022) have developed algorithms for maintaining a minimum spanning forest, which they used for graph clustering. Wang et al. (2021) developed a batch-parallel data structure for solving the closest pair problem.

There has also been significant work on batch-parallel graph algorithms in the Massively Parallel Computation (MPC) model. For example, Italiano et al. (2019) show how to process batch updates to graphs, while supporting connectivity queries and while maintaining a minimum spanning tree or a matching. Gilbert and Li (2020) generalise this for the k-server model, a different model of parallel computation. Dhulipala et al. (2020) provide yet more general results for maintaining dynamic graphs subject to batch updates in the MPC model.

Batcher. The design of OBatcher is inspired by Agrawal et al.'s Batcher (2014)—a scheduler implemented as a part of modified Cilk-5 runtime thats allows to make batching implicit in a way that is transparent to the user. Rather than being a lightweight solution based on provided programming language mechanisms, Batcher fundamentally is an extension of a standard randomised work-stealing scheduler that allows operations to a data structure to implicitly form a batch before being sent for processing in an asynchronous setting. For context, it is important to note that the main contribution of Agrawal et al.'s work is primarily on a theoretical side: their aim is to provide a scheduler with a provable complexity bound on an execution time of a program that manipulates a batch-parallel data structure, in terms of its *work* (the total number of operations the program must execute) and *span* (the length of the longest non-parallelisable sequence of operations). The statement of such a performance theorem, if possible, would be further complicated in the case of multiple concurrently used batch-parallel data structures, hence Batcher only supports one such structure. In contrast, OBatcher supports multiple data structures by design, but comes with no formal guarantees regarding its performance.

Concurrent search structures. In this work, we have implemented a variety of concurrent search structures. For some of these data structures, such as B-trees, Red-Black trees and AVL trees, there is a long history of concurrent implementations (Bayer and Schkolnick 1977; Ellis 1980), as well as more recent work, particularly on B-trees and related structures (Braginsky and Petrank 2012; Srivastava and Brown 2022). Most relevant to this paper is the work by Blelloch et al. (2016, 2022); Blelloch and Reid-Miller (1998), who showed how splitting and joining trees yielded very efficient parallel solutions—our implementation of the split-join paradigm followed this approach.

Search structures with sub-logarithmic time, such as the van Emde Boas tree, x-fast tries, and y-fast tries, have been much more challenging to implement in concurrent settings, and there exist relatively few such parallel/concurrent implementations. Of note, the only concurrent implementation (to our knowledge) of an x-fast (or y-fast) trie is the SkipTrie by Oshman and Shavit (2013). The first parallel version of the van Emde Boas tree (to the best of our knowledge) was just published by Gu et al. (2023). More recently, Khalaji et al. (2024) developed a very impressive concurrent van Emde Boas tree implementation relying on Intel's Hardware Transactional Memory instructions. A related sub-logarithmic data structure was developed by Brown et al. (2020), providing a very efficient concurrent implementation of an Interpolation Tree, not considered in this paper.

7 Conclusion

In this work, we have brought batch-parallelism to Multicore OCaml by introducing OBatcher, a library providing a lightweight embedding of implicit batching and a toolkit for constructing batch-parallel structures. The key observation behind our library is that the inversion of control provided by simple `async`/`await` asynchronous programming primitives is entirely sufficient to transform explicitly-batched interfaces into an implicitly-batched ones; we have demonstrated that as such OBatcher can easily be ported to other modern programming languages, such as Rust. Beyond this embedding, OBatcher investigates a methodology for gradually incorporating batch-parallelism into an existing codebases and structures, presenting, amongst which, two general patterns for decomposing the parallelisation of common search-structures, and instantiating these for widely-used structures such as X-fast and Y-fast tries, and van Emde Boas trees. Our experimental results show the resulting concurrent data-structures constructed using the OBatcher framework far outperform their coarse-grained counterparts and scale more gracefully as parallelism increases.

Inspired by this technique, in the future, we are planning to explore the possibilities it presents for automatically synthesising efficient parallel data structures that are correct by construction.

Acknowledgments

We are grateful to Arthur Wendling who hinted the initial design of OBatcher in Multicore OCaml. We thank Matthew Flatt and George Pîrlea for their feedback on earlier drafts of this paper. We also thank the anonymous reviewers of ICFP'23 and OOPSLA'24 for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

Data Availability

The software artefact accompanying this paper is available online (Le et al. 2024b). The artefact contains the source code of OBatcher (in OCaml and Rust), implementations of all data structures described in Sec. 4, and build scripts for reproducing the evaluation results from Sec. 5.

References

Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-Dynamic Trees via Change Propagation. In *ESA (LIPIcs, Vol. 173)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:23. https://doi.org/10.4230/LIPIcs.ESA.2020.2

Stephen Adams. 1993. Functional Pearls Efficient sets—a balancing act. *Journal of Functional Programming* 3, 4 (1993), 553–561. https://doi.org/10.1017/S0956796800000885

Georgy Adelson-Velsky and Evgenii Landis. 1962. An algorithm for the organization of information. *Proc. of the USSR Academy of Sciences* 145 (1962), 263–266. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962.

Kunal Agrawal, 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, 389–390. https://doi.org/10.1145/2555243.2555284

Yaroslav Akhremtsev and Peter Sanders. 2016. Fast Parallel Operations on Search Trees. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 291–300. https://doi.org/10.1109/HiPC.2016.042

Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica* 1 (1972), 290–306. https://doi.org/10.1007/BF00289509

Rudolf Bayer and Edward M McCreight. 1972. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. In *Acta informatica*, Vol. 1. Springer, 290–306. https://doi.org/10.1007/BF00289509

Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977), 1–21. https://doi.org/10.1007/BF00263762

Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *SPAA*. ACM, 253–264. https://doi.org/10.1145/2935764.2935768

Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2 (2022), 7:1–7:41. https://doi.org/10.1145/3512769

Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *SPAA*. ACM, 16–26. https://doi.org/10.1145/277651.277660

Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+tree. In *SPAA*. ACM, 58–67. https://doi.org/10.1145/2312005.2312016

Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. 1998. A Parallel Priority Queue with Constant Time Operations. *J. Parallel Distributed Comput.* 49, 1 (1998), 4–21. https://doi.org/10.1006/jpdc.1998.1425

Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *PPoPP*. ACM, 276–291. https://doi.org/10.1145/3332466.3374542

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

Simon Cruanes. 2024. *An in-memory Datalog implementation for OCaml.* Available at https://github.com/c-cube/datalog, accessed on 28 March 2024.

Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds. In *SODA*. SIAM, 1300–1319. https://doi.org/10.1137/1.9781611975994.79

James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. 1988. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Commun. ACM* 31, 11 (1988), 1343–1354. https://doi.org/10.1145/50087.50096

Carla Schlatter Ellis. 1980. Concurrent Search and Insertion in AVL Trees. *IEEE Trans. Computers* 29, 9 (1980), 811–817. https://doi.org/10.1109/TC.1980.1675680

Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *SPAA*. ACM, 325–334. https://doi.org/10.1145/1989493.1989549

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 128:1–128:29. https://doi.org/10.1145/3428196

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. ACM, 212–223. https://doi.org/10.1145/277650.277725

Seth Gilbert and Lawrence Er Lu Li. 2020. How Fast Can You Update Your MST?. In *SPAA*. ACM, 531–533. https://doi.org/10.1145/3350755.3400240

Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *SPAA*. ACM, 327–340. https://doi.org/10.1145/3558481.3591069

Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. 2023. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS. In *OSDI*. USENIX Association, 1–16. https://www.usenix.org/conference/osdi23/presentation/gupta

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *SPAA*. ACM, 355–364. https://doi.org/10.1145/1810479.1810540

Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *SIROCCO (LNCS, Vol. 4474)*. Springer, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann. https://dl.acm.org/doi/book/10.5555/2385452

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. 2019. Dynamic Algorithms for the Massively Parallel Computation Model. In *SPAA*. ACM, 49–58. https://doi.org/10.1145/3323165.3323202

Mohammad Khalaji, Trevor Brown, Khuzaima Daudjee, and Vitaly Aksenov. 2024. Practical Hardware Transactional vEB Trees. In *PPoPP*. ACM, 215–228. https://doi.org/10.1145/3627535.3638504

Callista Le, Kiran Gopinathan, Koon Wen Lee, Seth Gilbert, and Ilya Sergey. 2024a. Concurrent Data Structures Made Easy (Extended Version). *CoRR* abs/2408.13779 (2024). https://doi.org/10.48550/arXiv.2408.13779

Callista Le, Kiran Gopinathan, Koon Wen Lee, Seth Gilbert, and Ilya Sergey. 2024b. *OBatcher: Implementation, Data Structures, and Experiments (OOPSLA'24 Artefact)*. https://doi.org/10.5281/zenodo.12604575

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2016. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.* 33, 4 (2016), 13:1–13:62. https://dl.acm.org/citation.cfm?id=2845079

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A concurrent program logic with a future and history. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1378–1407. https://doi.org/10.1145/3563337

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI*. ACM, 809–824. https://doi.org/10.1145/3519939.3523432

Rotem Oshman and Nir Shavit. 2013. The SkipTrie: low-depth concurrent search without rebalancing. In *PODC*, Panagiota Fatourou and Gadi Taubenfeld (Eds.). ACM, 23–32. https://doi.org/10.1145/2484239.2484270

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 2000. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*. World Scientific, 182–204.

Richard C. Paige and Clyde P. Kruskal. 1985. Parallel Algorithms for Shortest Path Problems. In *ICPP*. IEEE Computer Society Press, 14–20.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS (LIPIcs, Vol. 2)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 383–414. https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1769

Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Euro-Par (LNCS, Vol. 6853)*. Springer, 136–147. https://doi.org/10.1007/978-3-642-23397-5_14

William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977

Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *SOSP*. ACM, 342–358. https://doi.org/10.1145/3132747.3132771

Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. 2019. *Sorted Sequences*. Springer International Publishing, Cham, 233–258. https://doi.org/10.1007/978-3-030-25209-0_7

Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (1996), 464–497. https://doi.org/10.1007/BF01940876

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-Grained Concurrent Programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221. https://doi.org/10.1145/3453483.3454039

Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-trees with fast, durable updates. In *PPoPP*. ACM, 416–430. https://doi.org/10.1145/3503221.3508441

R. Kent Treiber. 1986. *Systems programming: coping with parallelism*. Technical Report RJ 5118. IBM Almaden Research Center.

Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 92–106. https://doi.org/10.1137/1.9781611975499.8

Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic Agglomerative Graph Clustering. In *SPAA*. ACM, 233–245. https://doi.org/10.1145/3490148.3538584

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Ph. D. Dissertation. University of Cambridge.

Peter van Emde Boas. 1975. Preserving Order in a Forest in less than Logarithmic Time. In *FOCS*. IEEE Computer Society, 75–84. https://doi.org/10.1109/SFCS.1975.26

Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In *37th International Symposium on Computational Geometry (SoCG 2021) (LIPIcs, Vol. 189)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 60:1–60:16. https://doi.org/10.4230/LIPIcs.SoCG.2021.60

Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space O(N). *Inform. Process. Lett.* 17, 2 (1983), 81–84. https://doi.org/10.1016/0020-0190(83)90075-3