YaleNUSCollege

Simple and Efficient Concurrent Data Structures via Batch Parallelism

Le Nguyen Phong

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Dr. Ilya Sergey

AY 2023/2024

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

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

✓ Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

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

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

Nguyen Phong Le, Cendana College
Name & Residential College of Student

_____ 07/04/2024
Signature of Student Date

Ilya Sergey _____ 7 April 2024
Name & Signature of Supervisor Date

Acknowledgements

I would like to first and foremost extend my heartfelt gratitude to Prof. Ilya Sergey. I am grateful for his wisdom and unending patience during this capstone project and beyond. He initiated me to the world of concurrent and parallel programming during my third year, and has been guiding me ever since.

To Prof. Seth Gilbert and Kiran, thank you for your valuable input and help throughout this past year. Your wealth of knowledge and experience never ceased to astound me, and this project would not have come to fruition without you.

To my RCA mates - Dylan, Ethan, Nihal, Katif, Jesse, Julene, Wei Lin, Angeline, Danan, Sherrill, Yejin, and Mayuko. You made Yale-NUS College feel like home and accepted me in full for who I am. No words can describe how grateful I am to have met all of you.

And to all of my friends at Yale-NUS College, thank you for making my four years of college such a memorable experience.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Simple and Efficient Concurrent Data Structures via Batch Parallelism

by Nguyen Phong LE

Designing a concurrent data structure can be challenging. Usually, the choice is between *coarse-grained concurrency*, which is simple to design and implement but does not perform well with higher numbers of threads, and *fine-grained concurrency*, which promises better performance but is notoriously difficult to implement. In this paper, we propose a systematic methodology to derive simple yet performant concurrent data structure based of the concept of batch parallelism - processing operations in batches and parallelising operations within a batch. Our principal contributions are two batch parallelisation strategies, rendered as OCaml functors, along with detailed case studies and evaluations of sequential data structures made concurrent using these strategies.

Contents

List of Figures

Chapter 1

Introduction

"Send help."

A student taking PCDP, probably

1.1 Concurrent Data Structures are Hard

Multicore processors are extremely common today, allowing multiple computing workloads to be spread out as *threads* among many cores and executed in parallel (*multiprocessing*), thus shortening overall computation time relative to simply running them sequentially on a single core. However, it is rare that these workloads are entirely independent of each other. Indeed, in most multi-threaded applications, they frequently access and update shared data, and these reads and writes need to be properly synchronised to avoid problems like data races that can result in loss of data or even an invalid state for the data structure in question.

Therefore, designing a thread-safe, concurrent data structure is crucial

for taking advantage of multiprocessing. Given a sequential data structure, we can take one of two routes to create a concurrent implementation. One is *coarse-grained concurrency*, which involves using a lock to ensure mutually exclusive access to a data structure. This means that only one thread can acquire the lock and access the underlying sequential data structure at any given time. This approach comes with the advantage of simplicity, as the process for implementing coarse-grained concurrency is nearly the same for all data structures and is thus largely mechanical. However, this also means that all operations involving a coarse-grained data structure are effectively sequential, and we do not get any speed-up from parallel processing *within* the data structure itself. This approach can also result in slowdowns for higher numbers of cores as lock contention increases and more parallel threads compete for a single lock.

The other approach is *fine-grained concurrency*, where we discard the global lock in favour of a more granular design, where various operations on the data structures are carefully considered and implemented such that, where they might overlap, the end result remains correct. This can mean, for example, a greater reliance on atomic operations like compare and swap, and only using locks sparingly and on select portions of the data structure. This allows us to take advantage of multiprocessing to execute operations on the data structures in parallel, resulting in better performance and scaling for higher numbers of cores. However, fine-grained concurrent data structures are notoriously hard to design and implement, due to their sheer complexity and the non-deterministic nature of data races that may arise (Herlihy and Shavit, 2008).

There is space then for an approach that takes the best of both worlds

- a way to create concurrent data structures that enable parallel operations while not incurring the design cost of a full-blown fine-grained data structure. *Batch parallelism* promises to be one such approach.

1.2 Batch Parallelism to the Rescue

Like its name suggests, batch-parallel data structures process operations in batches. The intuition behind such an approach is that it is easier to parallelise a *batch of known operations* than random operations as they arrive. However, creating such an explicit batch of operations is highly inconvenient and greatly complicates the interface of the data structure, as the onus is on the client to properly gather such operations into a single batch. We call this approach *explicit batching*.

To solve this problem, Agrawal et al. (2014) proposed *implicit batching*, where the batching of operations is abstracted away from the user by means of a custom scheduler, leaving only a direct interface that can be used to interact with the batch-parallel data structure just like any other concurrent data structure. However, as presented, this method requires invasive modification of the implementation language's runtime scheduler, which can hardly be called trivial, and can only accomodate a single data structure per application, making its usefulness limited.

More recently, Lee (2023) proposed a way to implement implicit batching using common concurrent progamming primitives, namely `async` and `await` - scheduling a task asynchronously, then wait for its result elsewhere. They presented OBatcher, a library for Multicore OCaml that provides an interface for implementing multiple batch-parallel structures per application, without the need for modifying the runtime scheduler.

However, there is still no set blueprint for implementing batch-parallel data structures, and even with the interface provided by OBatcher, implementing one is still a less-than-trivial task. We propose in this paper a way to make this "batch parallelisation" more mechanical for certain classes of sequential data structures.

1.3 Contributions

More concretely, our contributions in this paper consist of the following:

1. Some improvements and fixes to the original OBatcher (Chapter 2).
2. Two patterns for implementing batch-parallel data structures, concretised as two ready-to-use OCaml functors for OBatcher. We first present the *split-join* functor, along with batch-parallel versions of the red-black tree, the AVL tree, and the treap implemented using it (Chapter 3). We follow this with the *expose-repair* functor, and batched parallel versions of the van Emde Boas tree, the x-fast trie, and the y-fast trie (Chapter 4).
3. A comprehensive evaluation of the performance of the above batch-parallel data structures, where we demonstrate that they perform favourably against their coarse-grained counterparts (Chapter 5).

All relevant code and usage instructions are available online.[1]

[1] https://github.com/phongulus/obatcher/tree/paper-artefact

Chapter 2

Background

2.1 A Quick Introduction to OBatcher

OBatcher is a new library for Multicore OCaml (OCaml 5), conceived and implemented by Lee (2023), with the goal of enabling users to instantiate implicitly batched data structures that can be interacted with via a simple direct style interface (e.g. a simple `apply t (Search k)` to search for some key `k` in the implicitly batched data structure `t`).

OBatcher is implemented on top of `Domainslib`, an OCaml library that provides support for nested parallel programming, and `await`/`async` primitives for creating parallel tasks and waiting for their results. These features are central to the implementation of OBatcher.

OBatcher provides the ability to create batched data stucture via the

```
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
```

FIGURE 2.1: OBatcher functor for direct-style structure.

functor shown in Figure 2.1. A functor in OCaml is essentially a parameterised module - in this example, given some explicitly batched data structure implemented as a module `S` of type `Batched`, the functor `Make` creates a new module containing the implicitly batched implementation with the types and functions in Figure 2.1.

The keen reader will have realised that given such an interface, the user will need to implement an explicitly batched version of the to-be-batch-parallelised data structure themselves. Even with the benefit of processing a batch of known operations rather than random operations, converting a sequential data structure into an explicitly batched one can still be a non-trivial task. We dedicate the bulk of this project to further simplifying this process by finding patterns for some data structures and implementing them as functors - but before that, we cover some more minor fixes and optimisations that we contributed to OBatcher.

2.2 Improving OBatcher

2.2.1 A Better Container

The type `t` in Figure 2.1 is essentially a wrapper extending the input data structure with additional constructs needed for implicit batching. As we can see from Figure 2.2, in addition to the underlying data structure, we have a `container` where pending operations are collected, an

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

FIGURE 2.2: Type `t` for implicitly batched structures.

atomic boolean `is_running` to check whether a batch of operations is currently being executed or not, and the thread pool `pool` provided by `Domainslib` for scheduling tasks. We will elaborate on `last_time` later.

Let us focus on `container`. It should be thread-safe and be able to accommodate multiple threads submitting and extracting operations to and from it. 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, used by the initial version of OBatcher.

However, we note, firstly, that we do not need the elements inside to be ordered - in other words, we do not care about which operations are submmitted first. Following the principle of *linearisability*, we can execute operations in any order, as long as the end result is as if we performed all our concurrent operations in some sequential order (Herlihy and Wing, 1990). Secondly, we will never need to dequeue anything less than all elements, as whenever we want to process a batch of operations, we can simply take every pending operations currently in the container.

Hence, our first improvement to OBatcher is replacing the current channel-based container with a 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. This version of the container removes the expensive list reversal logic for maintaining a first-in-first-out queue, and avoids the need to dequeue operations one by one from the container, thereby improving performance.

2.2.2 Minimum Batch Size and Timer

When a client task submits an operation to the container, a `try_launch` function is called to start execution of the batch. If launching the batch is sucessful, the task in question is effectively promoted to a worker that takes all elements currently present in the operations container and starts the parallel processing of the batched operations. Otherwise, the operation is submitted to the container and the task simply waits until a later batch is launched that would perform the submitted operation.

In its initial implementation, OBatcher will attempt to launch a batch as soon as it verifies that there is no batch currently being processed and that the number of pending operations in the container is non-zero. However, this can result in very small batch sizes and thus increased batcher overhead relative to the actual number of operations being performed.

Our second improvement to OBatcher is implementing a minimum batch size to avoid such small batch sizes. 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 - this timestamp is stored in the `last_run` field of the batched data structure type `t` (Figure 2.2).

Furthermore, 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.

2.2.3 Waiting for Batch Processing

Our last change to OBatcher is a fix to enable more accurate benchmarks. When we submit an operation to the batch, we also submit a *callback function* that would be invoked once the operation takes effect or when any subsequently submitted operation is *guaranteed* to take effect *after* that first operation. This callback function fulfills a *promise* that the submitting task is waiting on after sending the operation to the batcher. When processing a batch of insert operations, we invoke the callback function immediately as there is no need to wait for any return value, and any operation submitted next will be processed on the next batch run.

While this behaviour is correct under normal circumstances, in Lee (2023)'s OBatcher, this is slightly problematic for benchmarking as the main task running the benchmarking logic can resume and end the benchmark timer *before* the current batch of opearations is finished. To work around this, we supplement OBatcher with a `wait_for_batch` function that blocks the main task - the one with benchmarking logic - until all operations are finished.

2.3 Concurrent Data Structures at a Glance

We conclude this chapter with a brief overview of the state of concurrency research for the data structures that we are studying in this project: the red-black tree, the AVL tree, the treap, the van Emde Boas tree, the x-fast trie, and the y-fast trie.

The first three are variations of the *balanced binary tree*. Among them, the red-black tree and the AVL tree are especially well-known as standard

textbook data structures. Accordingly, there are already fine-grained concurrent implementations for them - for example, the concurrent implementation of AVL trees by Ellis (1980b), or the concurrent 2-3 tree (equivalent to a red-black tree), also by Ellis (1980a). We do not intend to compete with these fine-grained implementations in terms of performance, but rather show that through using our batching patterns, we can get a reasonable performance improvement over coarse-grained implementations at a fraction of the design cost.

The van Emde Boas tree, the x-fast trie, and the y-fast trie are sub-logarithmic search data structures, meaning that they enable queries in less than $O(\log n)$ time. These, while present in some textbooks (Cormen et al., 2009), are not as widely used and less work have been done to port them to a concurrent setting. To our knowledge, the closest known concurrent data structure to an x-fast or y-fast trie is the SkipTrie by Oshman and Shavit (2013). The van Emde Boas trie, meanwhile, only recently received its first concurrent implementation by Gu et al. (2023) - again, to the best of our knowledge.

Given the state of affairs, we can see the challenge with properly implementing concurrent, fine-grained versions of the above data structures. Therefore, our primary motivation is to show that it is possible to go around all this complexity and systematically derive batch-parallel versions from the original sequential data structure, with reasonably good performance. One of this work's key observations is that, for certain classes of sequential search structures, it is possible to identify implementation strategies that streamline the development of their batch-parallel

counterparts. In the following sections, we showcase two such strategies, embodied by OCaml functors: so-called *split-join* (Chapter 3) for the red-black tree, the AVL tree, and the treap; and *expose-repair* (Chapter 4) for the van Emde Boas tree, the x-fast trie, and the y-fast trie. We note that the case studies we describe here are limited to search data structures that implement maps and ordered sets, and we do not claim that our implementations are optimal in the theoretical sense. We leave such proofs for future work.

Chapter 3

Batching with Split-Join Strategy

The first batch parallelism strategy we explore is based on the idea of *splitting* and *joining* by Blelloch, Ferizovic, and Sun (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 crux of the approach is in splitting a tree instance in a way that 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 can join trees pair-wise where the maximum key element in the first tree, if any, is strictly less than the

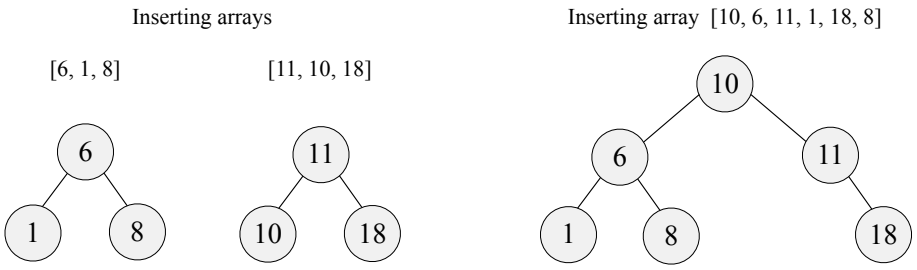


FIGURE 3.1: Example of splitting (left) and joining (right) a binary tree.

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 Figure 3.1, where most of the subtree structure remains unchanged. To achieve that, we adopt ideas from the work of Blelloch, Ferizovic, and Sun (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.

3.1 A Functor for Split-Join Batching

To provide a convenient abstract interface for split/join batch parallelism, we define two modules signatures, `Sequential` (Figure 3.2) and `Prebatch`

```
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
```

FIGURE 3.2: The `Sequential` module signature for the split-join functor.

(Figure 3.3), 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, and the query and update operations `search`, `insert`, and `delete`. Note that update operations 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 implement `Sequential` by defining all basic sequential operations that can be used to interface with the data structure. Should there be no need for concurrency, 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`.

```
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
```

FIGURE 3.3: The `Prebatch` module signature for the split-join functor.

- `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 a number that approximates the size of the current data structure. For instance, this can be the height of a tree. We will describe its use shortly.
- `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.

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. Let us examine its

```
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)
```

FIGURE 3.4: Split-join parallel batching.

`run_batch` function define in Figure 3.4. 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 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 in Section 2.2.1. 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 slightly 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

```
let par_insert ~pool s inserts =
  let n =
    Array.length inserts / seq_threshold + 1 in
  (* assume that inserts are randomly ordered *)
  let pivots =
    Array.init n (fun i -> fst inserts.(i)) in
  sort pivots;
  let s_arr = split_multiple s pivots in
  let sub_ranges =
    partition pivots inserts in
  parallel_for pool
    ~start:0
    ~finish:(Array.length sub_ranges)
    ~body:(fun i ->
      let range = sub_ranges.(i)
      for j = fst range to snd range do
        let k, v = inserts.(j) in
        S.insert s_arr.(i) k v
      done);
  set_root t (join_multiple s_arr)
```

FIGURE 3.5: Batch-parallel insertion for split-join.

`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 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 Figure 3.5. 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 (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.

3.2 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.

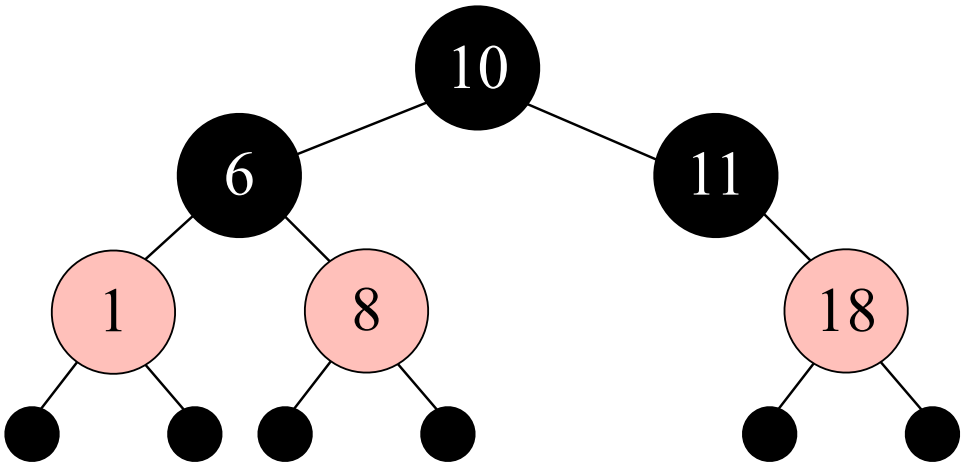


FIGURE 3.6: Example of a red-black tree.

3.2.1 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 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. Aside from this omission, our implementation follows the red-black tree algorithms as described by Cormen et al. (2009). Figure 3.6 shows the balanced binary tree that we have previously seen in Figure 3.1, but as a red-black tree, with its nodes coloured in black and red, and respecting the tree invariants stated above.

Following the `Sequential` signature from Figure 3.2, 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 and adding a new node at the leaf level, after which the tree can 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).

3.2.2 Batch Parallel Red-Black Tree Overview

The logic of a batch-parallel red-black tree follows the general split-join strategy from Section 3.1, 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, Ferizovic, and Sun, 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, Ferizovic, and Sun (2016), 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.

3.3 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. Blelloch, Ferizovic, and Sun (2016) also provide 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 implement their batch-parallel versions just like with the RB tree by defining their `split` and `join` functions as described by Blelloch, Ferizovic, and Sun (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.

Chapter 4

Batching with Expose-Repair Strategy

The split-join strategy discussed in Chapter 3 is beneficial for 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 data structure lends itself well to being batch-parallelised this way, meaning that they cannot be readily split into valid sub-structures, 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 a predictable, *localised* area that is the same regardless of what other keys populate the data structure.

However, batch parallelising such "position-based" search structures might not be as straightforward as it seems. Advanced examples like the van Emde Boas tree (van Emde Boas, 1977), 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 sequential operations. Those sub-structures must be accounted for, restored, and/or updated both before and after running a batch.

We present a novel strategy for batch parallelism, dubbed *expose-repair*, aimed at addressing such data structures. It centres around first "exposing", or preparing the structure before each batch of operations, so as to make sure parallel operations in their respective localised areas do not affect each other. Then, we "repair" the result after processing the batch of operations, to re-establish the global metadata or sub-structures.

4.1 Expose-Repair Functor Overview

Similarly to the split-join functor from Chapter 3, 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 store, 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 restored by associating satellite data with the keys. As the result of this design, the `Sequential` signature shown in Figure 4.1 offers the `mem` function instead of `search`. The signature also features the `predecessor` and `successor` functions, as one of the main selling points

```
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
```

FIGURE 4.1: The `Sequential` module signature for the expose-repair functor.

```
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
```

FIGURE 4.2: The `Prebatch` module signature for the expose-repair functor.

of these fast search structures, such as van Emde Boas tree, the x-fast and the y-fast tries, is 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 that makes use of the `Prebatch` functions. As a characteristic example,

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

FIGURE 4.3: Batch-parallel insertion in the expose-repair functor.

consider the implementation of parallel insertion via the expose-repair strategy shown in Figure 4.3.

We start by checking if the number of insert operations in the batch exceeds the sequential threshold, and if it is below, the operations are performed sequentially (we omit this part from the listing for brevity); otherwise, we continue with batch processing. To do so, we first obtain `n` sorted random "tentative pivots" from the batch of operations, and we transform 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 (Section 3.1), 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. 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 possible keys), so that we can divide up a cluster of trees without dissecting any tree in the middle (see Section 4.2).

Next, we prepare the data structure for batch processing where needed and partition 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.

Then, the sub-batches of operations resulting from partitioning are run in parallel on the same pre-processed data structure (lines 9-10). It is done with the assumption that `expose` has indeed returned the pivots in a way that split the range of the insertions such that operations in different sub-batches would not interfere with each other.

Finally, we repair the initial data structure `t` by, e.g., removing unused pre-initialised parts, updating metadata, and restoring auxiliary pointers between its sub-parts (line 11).

The code in Figure 4.3 relies on the following components of the `Prebatch` functor from Figure 4.2:

- `dt`: an abstract type specific for each data structure. It is used to store supplementary information for batching, if needed (see Section 4.2 for a concrete example).
- `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.

- `repair`: repairs the data structure after all updates have finished.
- `insert_range`: performs a sequence of insertions on 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.

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 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, 1977). 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.

4.2.1 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 vEB tree of universe size $\sqrt{u}$ so that a vEB tree of some 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.

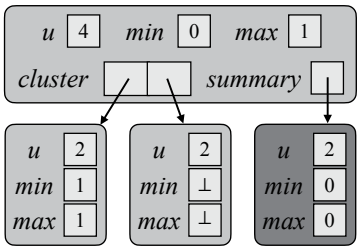


FIGURE 4.4: A vEB tree containing $\{0, 1\}$.

Consider the example vEB tree in Figure 4.4. 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). Conversely, the *maximum* field of the root vEB node shows 1, but this is *not* the key value itself, and is just metadata. Looking down, the left child node, corresponding to the index 0 of the root cluster, 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 the detailed descriptions of sequential vEB operations, and refer the reader to Chapter 20 of the textbook by Cormen et al. (2009).

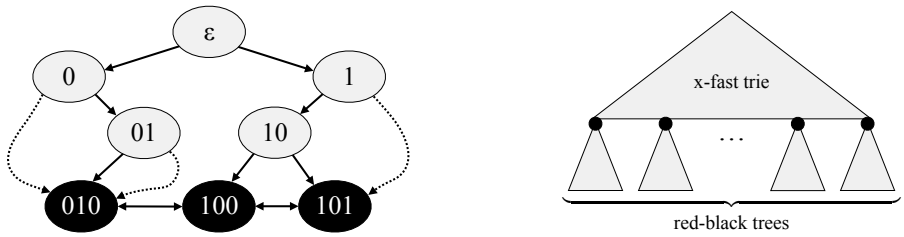
4.2.2 Batch Parallel van Emde Boas Tree Overview

The vEB tree lends itself naturally to batch processing. We note that since there are about $\sqrt{u}$ trees available in the cluster at the root level, we can have each domain handle multiple vEB trees from this cluster. Since there are already so many vEB trees at this first level, we will not look further below for parallelising our batched operations.

```
let expose t pivot_seeds =
  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, ())
```

FIGURE 4.5: The vEB tree `expose` function.

The code of the `expose` function for vEB tree is shown in Figure 4.5. 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. For this, we



(A) The x-fast trie (B) The y-fast trie

FIGURE 4.6: X-fast trie and Y-fast tries. For the x-fast trie in Figure 4.6a, solid double arrows indicate pointers from each leaf to the previous and the next leaf, while dashed single arrows indicate descendant pointers.

simply transform each pivot key into the minimum possible key for its target sub-vEB tree in the root-level cluster. This ensures that keys in different sub-batches are never inserted in the same vEB tree. At the end, we deduplicate the final list of pivots, as 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 `repair` function.

4.3 Other Expose-Repair Data Structures

In addition to the vEB tree, we have instantiated the expose-repair functor 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 (Figure 4.6a), 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 to the largest leaf in its left subtree; in both cases those are referred to as *descendant* pointers. At each layer, there is a hash table to accelerate queries. Unfortunately, this last point complicates batch-parallelising, as it would require a concurrent hash table. 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 (Figure 4.6b), we replace the x-fast trie's leaf nodes with a forest of red-black (RB) 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 either. 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. Operations within each range are dispatched sequentially by `insert_range`. Finally, `repair` checks the size of each resulting RB tree, splitting where needed to maintain the size bound.

Chapter 5

Experiments

In this section, we provide an extensive evaluation of the throughput (operations per second) trends of the 6 batch-parallel search data structures presented thus far. 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 and scaling with less work.

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

We evaluate the throughput for all 6 search structures (RB tree, AVL tree, treap, van Emde Boas tree, x-fast and y-fast tries) 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. We summarise our observations from these experiments in this chapter, and refer the reader to Appendix A for the benchmark results.

5.1 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 (Chapter 2), due to not meeting the minimum batch size—as there are no other threads contributing operations. We conclude that, without further optimization, 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.

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.

5.2 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 surmise that this is a rare case where the underlying operation is so fast that the lock contention imposes a smaller overall overhead than starting a batch.

5.3 Comparison With a Fine-Grained Skip List

We end our evaluation with a quick discussion of the batch parallel skip list's performance versus that of the fine-grained skip list. Lee (2023) previously ported an implementation the fine-grained skip list by Herlihy et al. (2007) from Java to OCaml for their initial work on OBatcher. Here, we benchmark it once more in Figure A.7 against Lee (2023)'s batch-parallel implementation of the skip list, using the same setup as above. Sadly, our results clearly indicate the growing performance gap between the fine-grained and the batch-parallel implementation.

We note however that the fine-grained implementation is significantly more intricate than the batch-parallel version (Lee, 2023), hence we are making a trade-off between performance and design complexity. Furthermore, we note that the current batch-parallel skip list does not make use of the split-join functor nor the expose-repair functor, but is rather implemented in an ad-hoc manner directly on top of OBatcher's batching interface. Further work can be done with regards to either batch the skip list in such a way that it follows these batching patterns, or to further generalise the split-join and/or expose-repair functors to accomodate a batch-parallel skip list. Doing so would significantly lighten the burden of designing a concurrent skip list.

Chapter 6

Discussion and Future Work

6.1 Bechmarking Matters

Overall, our benchmark results showed that there is much to be gained by using OBatcher along with either the split-join or the expose-repair functors for generating concurrent data structures. The performance scaling is not very good for higher core counts beyond 8. It is worth pointing out here a quirk with how we engineered the benchmarks. While operations are simply run one after another in the sequential benchmark, they are submitted as *individual tasks* to the `Domainslib` task queue for the batch-parallel benchmark. This means that threads must each concurrently dequeue one operation, then submit it to the batch, then repeat until all operations are submitted. This adds significant overhead and puts the batch-parallel data structures at a disadvantage relative to their sequential counterparts for the benchmark - meaning that it is all the more impressive how for some workloads and data structures, the batch-parallel implementation actually matches or even outperform the sequential implementation. Therefore, looking into more benchmarking

methods or using closer-to-real-world workloads may prove to be fruitful in showcasing the full potential of batch-parallel data structures.

6.2 Split-Join and Expose-Repair Functors

There is potential for more functionality and performance within the split-join and expose-repair functors.

As presented, our functors currently lack some functionalities, such as support for batched delete operations. However, these should be fairly trivial to implement following the model of how batched inserts are handled. The expose-repair functor can also be expanded to support storing key-value pairs - this feature was ommitted since the van Emde Boas tree, the x-fast trie, and the y-fast trie do not traditionally support it.

We also do not make any claim about whether the batch processing methods that we used for the data structures are optimal. For instance, in the split-join functor, we opted for splitting the data structure all at once in the beginning to obtain an array of split data structures. Another approach that we experimented with was *breaking apart* the root node, and spawning two parallel tasks that do the same for the sub-trees in a recursive manner until a certain level is reached. Though we ultimately went with the approach described in Chapter 3 thanks to it being potentially generalisable to more than just binary trees (e.g. it does not rely on there being a *node* exposed to the client), it is far from the only way to parallelise a batch of operations. Finding better ways to process a batch may be a topic for future research.

Chapter 7

Conclusion

To conclude, we have shown in this paper that some data structures can be easily batch-parallelised following two patterns: split-join and expose-repair. We have implemented these patterns as OCaml functors for the OBatcher library, and demonstrated their use by implementing a host of data structures: the red-black tree, the AVL tree, the treap, the van Emde Boas tree, the x-fast trie, and the y-fast trie. We have also demonstrated through benchmarks that our implementations exhibit reasonable performance relative to coarse-grained implementations, and is even competitive under certain circumstances with sequential implementations.

Bibliography

Adams, Stephen (1993). "Functional Pearls Efficient sets—a balancing act". In: *Journal of Functional Programming* 3.4, 553–561. DOI: 10.1017/S0956796800000885.

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

Agrawal, Kunal, Jeremy T. Fineman, Brendan Sheridan, Jim Sukha, and Robert Utterback (2014). "Provably Good Scheduling for Parallel Programs that Use Data Structures through Implicit Batching". In: *PPoPP*. ACM, pp. 389–390. DOI: 10.1145/2555243.2555284. URL: https://doi.org/10.1145/2555243.2555284.

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

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

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

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

Ellis, Carla Schlatter (1980a). "Concurrent search and insertion in 2—3 trees". In: *Acta Inf.* 14.1, 63–86. ISSN: 0001-5903. DOI: 10.1007/BF00289064. URL: https://doi.org/10.1007/BF00289064.

— (1980b). "Concurrent Search and Insertion in AVL Trees". In: *IEEE Trans. Computers* 29.9, pp. 811–817. DOI: 10.1109/TC.1980.1675680. URL: https://doi.org/10.1109/TC.1980.1675680.

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

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

Herlihy, Maurice and Nir Shavit (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann. ISBN: 978-0-12-370591-4.

Herlihy, Maurice and Jeannette M. Wing (1990). "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3, pp. 463–492. DOI: 10.1145/78969.78972.

Lee, Koon Wen (2023). "Concurrent Structures and Effect Handlers: A Batch Made in Heaven". Bachelor's (Hons.) Thesis. Yale-NUS College.

URL: https://ilyasergey.net/assets/pdf/papers/Koon-Wen-Lee-Capstone.pdf.

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

Sanders, Peter, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev (2019). "Sorted Sequences". In: *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Cham: Springer International Publishing, pp. 233–258. ISBN: 978-3-030-25209-0. DOI: 10.1007/978-3-030-25209-0_7. URL: https://doi.org/10.1007/978-3-030-25209-0_7.

Seidel, Raimund and Cecilia R. Aragon (1996). "Randomized Search Trees". In: *Algorithmica* 16.4/5, pp. 464–497. DOI: 10.1007/BF01940876. URL: https://doi.org/10.1007/BF01940876.

Treiber, R. Kent (1986). *Systems programming: coping with parallelism*. Tech. rep. RJ 5118. IBM Almaden Research Center.

van Emde Boas, Peter (1977). "Preserving order in a forest in less than logarithmic time and linear space". In: *Information Processing Letters* 6.3, pp. 80–82. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(77)90031-X. URL: https://www.sciencedirect.com/science/article/pii/002001907790031X.

Willard, Dan E. (1983). "Log-logarithmic worst-case range queries are possible in space O(N)". In: *Information Processing Letters* 17.2, pp. 81–84. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(83)90075-3.

Appendix A

Benchmark Results for Batch Parallel Data Structures

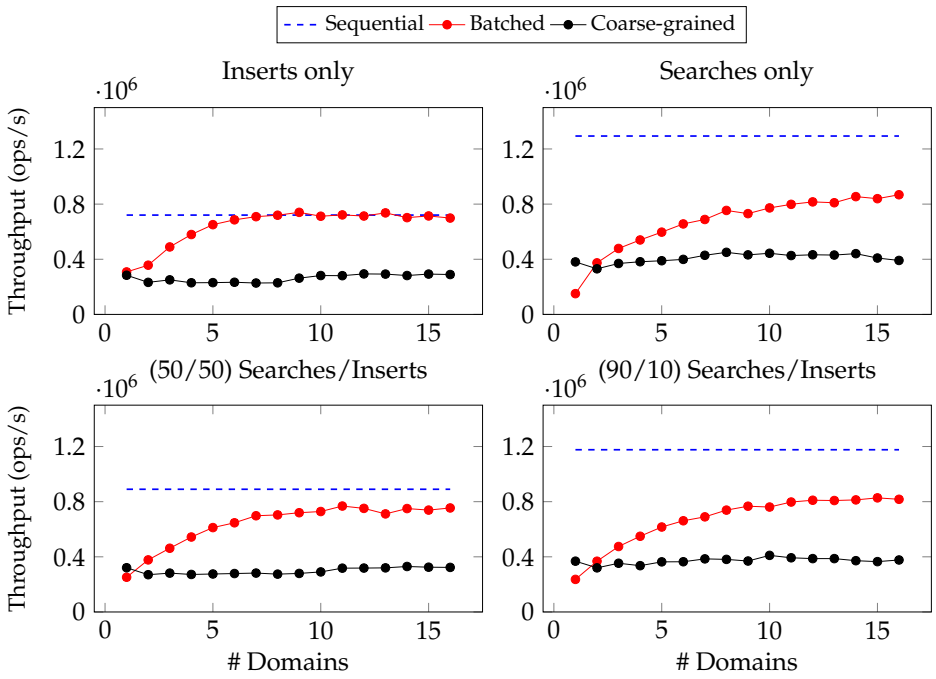


FIGURE A.1: Throughput comparison for the batch-parallel red-black tree from Chapter 3.

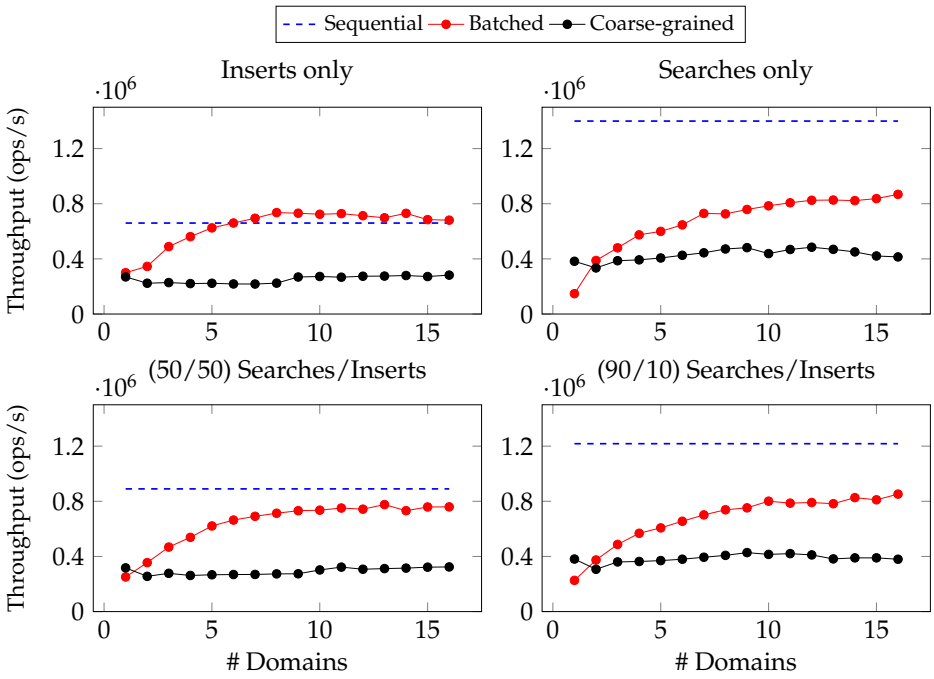


FIGURE A.2: Throughput comparison for the batch-parallel AVL tree from Chapter 3.

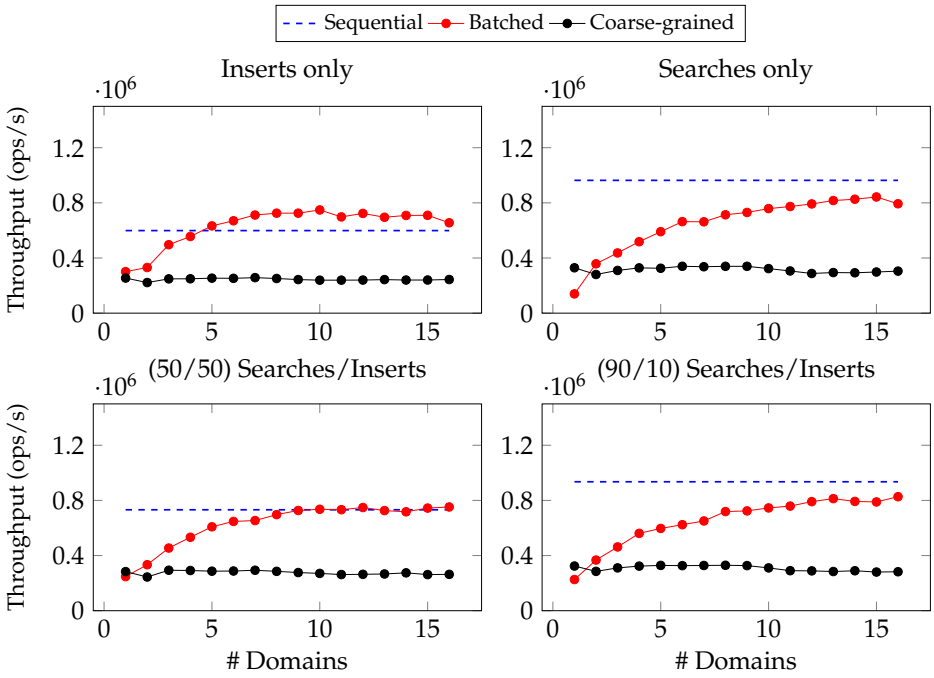


FIGURE A.3: Throughput comparison for the batch-parallel treap from Chapter 3.

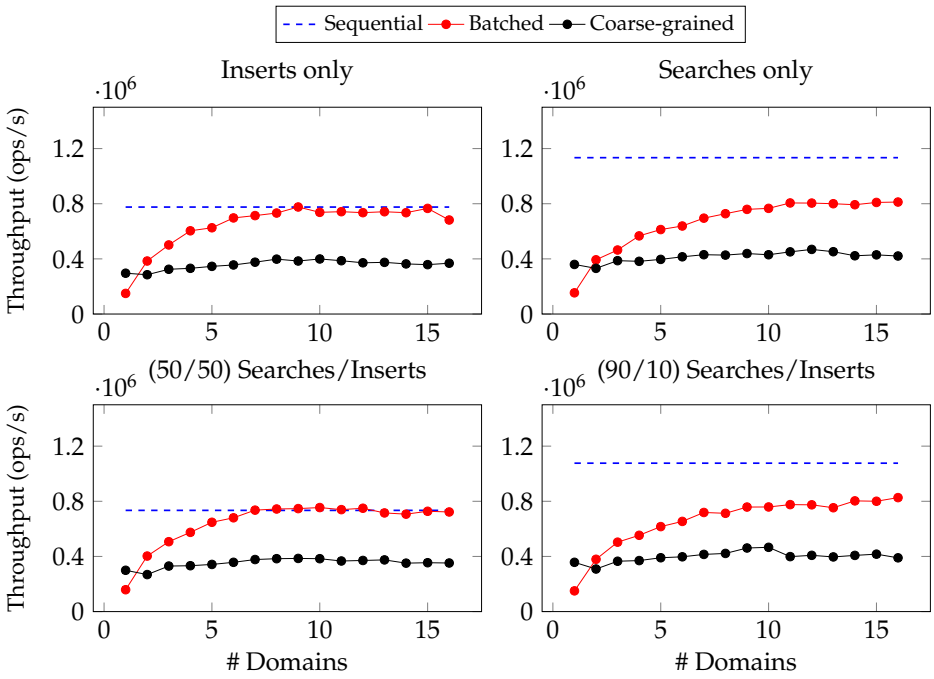


FIGURE A.4: Throughput comparison for the batch-parallel van Emde Boas tree from Chapter 4.

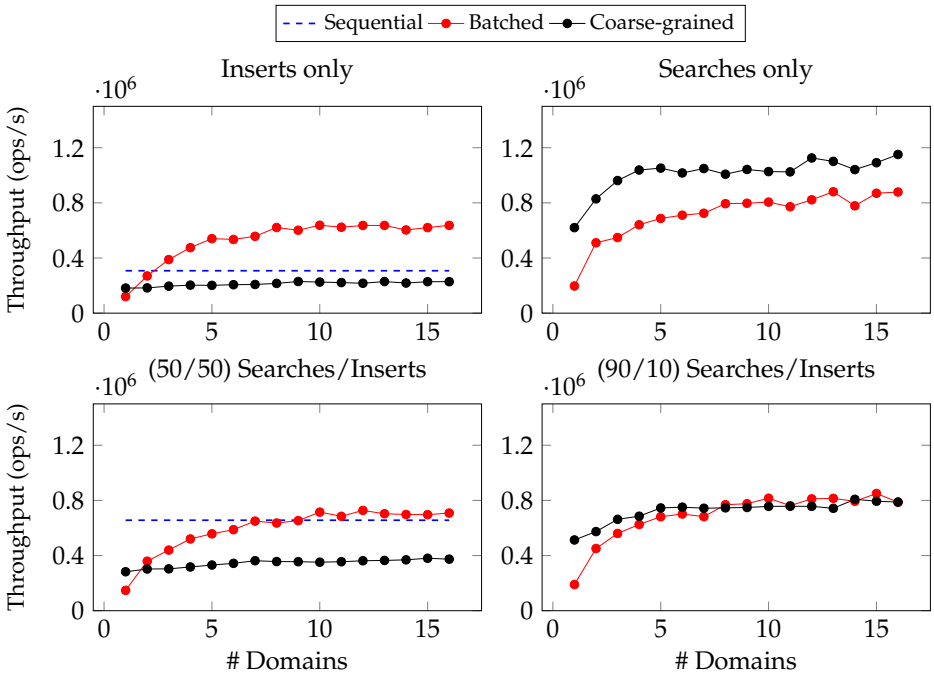


FIGURE A.5: Throughput comparison for the batch-parallel x-fast trie from Chapter 4.

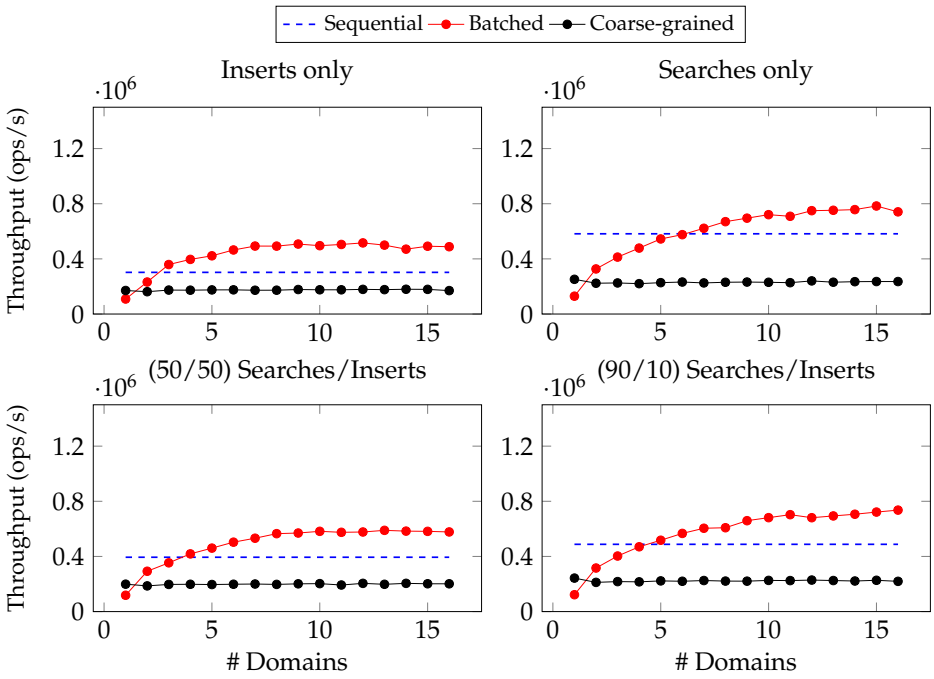


FIGURE A.6: Throughput comparison for the batch-parallel y-fast trie from Chapter 4.

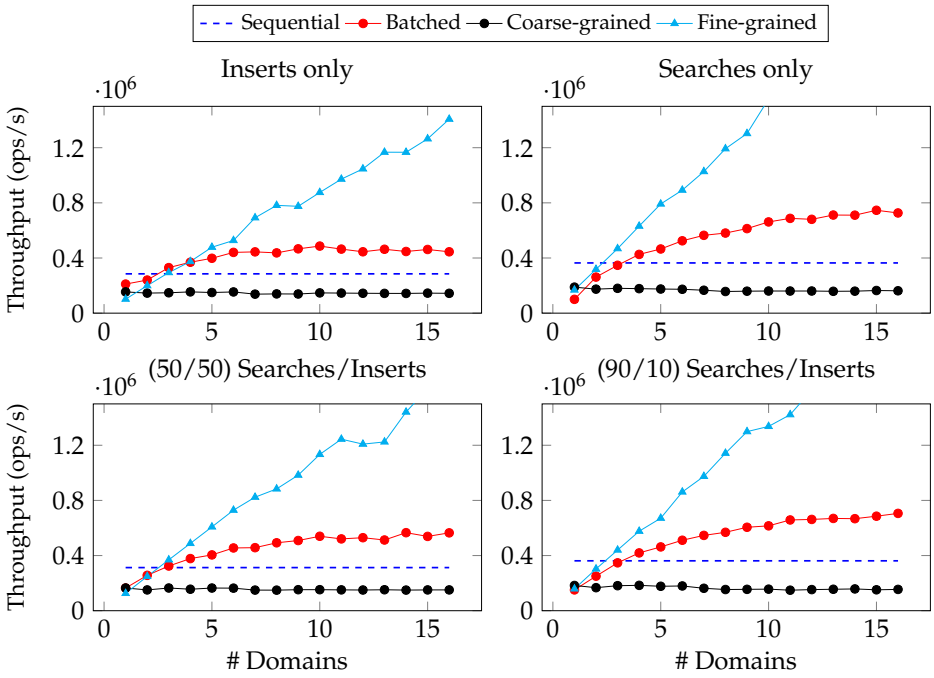


FIGURE A.7: Throughput comparison for the batch-parallel skip list.