Parallel Finger Search Structures

Seth Gilbert
Computer Science, National University of Singapore

Wei Quan Lim
Computer Science, National University of Singapore

Abstract

In this paper we present two versions of a **parallel finger structure** $\mathbb{FS}$ on p processors that supports searches, insertions and deletions, and has a finger at each end. This is to our knowledge the first implementation of a parallel search structure that is ***work-optimal*** with respect to the finger bound and yet has very good parallelism (within a factor of $O\big((\log p)^2\big)$ of optimal). We utilize an **extended implicit batching** framework that transparently facilitates the use of $\mathbb{FS}$ by any parallel program P that is modelled by a dynamically generated DAG D where each node is either a unit-time instruction or a call to $\mathbb{FS}$.

The work done by $\mathbb{FS}$ is bounded by the **finger bound** F_L (for some linearization L of D), i.e. each operation on an item with distance r from a finger takes $O(\log r + 1)$ amortized work. Running P using the simpler version takes $O\big(\frac{T_1+F_L}{p} + T_\infty + d \cdot \big((\log p)^2 + \log n\big)\big)$ time on a greedy scheduler, where T_1, T_∞ are the size and span of D respectively, and n is the maximum number of items in $\mathbb{FS}$, and d is the maximum number of calls to $\mathbb{FS}$ along any path in D. Using the faster version, this is reduced to $O\big(\frac{T_1+F_L}{p} + T_\infty + d \cdot (\log p)^2 + s_L\big)$ time, where s_L is the weighted span of D where each call to $\mathbb{FS}$ is weighted by its cost according to F_L. $\mathbb{FS}$ can be extended to a fixed number of movable fingers.

The data structures in our paper fit into the ***dynamic multithreading*** paradigm, and their performance bounds are directly ***composable*** with other data structures given in the same paradigm. Also, the results can be translated to practical implementations using work-stealing schedulers.

2012 ACM Subject Classification Theory of computation → Parallel algorithms; Theory of computation → Shared memory algorithms; Theory of computation → Parallel computing models

Keywords and phrases Parallel data structures, Multithreading, Dictionaries, Comparison-based Search, Distribution-sensitive algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.20

Related Version The full version of this paper is available at `https://arxiv.org/abs/1908.02741`.

Funding This research was supported in part by Singapore MOE AcRF Tier 1 grant T1 251RES1719.

Acknowledgements We would like to express our gratitude to our families and friends for their wholehearted support, to the kind reviewers who provided helpful feedback, and to all others who have given us valuable comments and advice.

1 Introduction

There has been much research on designing parallel programs and parallel data structures. The **dynamic multithreading paradigm** (see [12] chap. 27) is one common parallel programming model, in which algorithmic parallelism is expressed through parallel programming primitives such as fork/join (also spawn/sync), parallel loops and synchronized methods, but the program cannot stipulate any mapping from subcomputations to processors. This is the case with many parallel languages and libraries, such as Cilk dialects [18, 23], Intel TBB [28], Microsoft Task Parallel Library [30] and subsets of OpenMP [25].

Recently, Agrawal et al. [3] introduced the exciting ***modular design*** approach of **implicit batching**, in which the programmer writes a multithreaded parallel program that uses a ***black box*** data structure, treating calls to the data structure as basic operations, and also

provides a data structure that supports batched operations. Given these, the runtime system automatically combines these two components together, buffering data structure operations generated by the program, and executing them in batches on the data structure.

This idea was extended in [4] to data structures that do not process only one batch at a time. In this **extended implicit batching framework**, the runtime system not only holds the data structure operations in a **parallel buffer**, to form the next **batch**, but also **notifies** the data structure on receiving the first operation in each batch. Independently, the data structure can at any point **flush** the parallel buffer to get the next batch.

This framework nicely supports ***pipelined*** batched data structures, since the data structure can decide when it is ready to get the next input batch from the parallel buffer. Furthermore, this framework makes it easy for us to build ***composable*** parallel algorithms and data structures with composable performance bounds. This is demonstrated by both the parallel working-set map in [4] and the parallel finger structure in this paper.

Finger Structures

The **map** (or **dictionary**) data structure, which supports inserts, deletes and searches/updates, collectively referred to as **accesses**, comes in many different kinds. A common implementation of a map is a balanced binary search tree such as an AVL tree or a red-black tree, which (in the comparison model) takes $O(\log n)$ worst-case cost per access for a tree with n items. There are also maps such as splay trees [29] that have amortized rather than worst-case performance bounds.

A **finger structure** is a special kind of map that comes with a **fixed finger** at each end and a (fixed) number of **movable fingers**, each of which has a key (possibly $-\infty$ or ∞ or between adjacent items in the map) that determines its position in the map, such that accessing items nearer the fingers is cheaper. For instance, the finger tree [20] was designed to have the finger property in the worst case; it takes $O(\log r + 1)$ steps per operation with finger distance r (Definition 1), so its total cost satisfies the finger bound (Definition 2).

▶ **Definition 1** (Finger Distance). *Define the **finger distance of accessing an item** x on a finger structure M to be the number of items from x to the nearest finger in M (including x), and the **finger distance of moving a finger** to be the distance moved.*

▶ **Definition 2** (Finger Bound). *Given any sequence L of N operations on a finger structure M, let F_L denote the **finger bound** for L, defined by $F_L = \sum_{i=1}^{N}(\log r_i + 1)$ where r_i is the finger distance of the i-th operation in L when L is performed on M.*

Main Results

We present, to the best of our knowledge, the first parallel finger structure. In particular, we design two parallel maps that are ***work-optimal*** with respect to the Finger Bound F_L (i.e. it takes $O(F_L)$ work) for some linearization L of the operations (that is consistent with the results), while having very good parallelism. (We assume that each key comparison takes $O(1)$ steps.) In this paper we focus on basic finger structures with just one fixed finger at each end (no movable fingers).

These parallel finger structures can be used by any parallel program P, whose actual execution is captured by a **program DAG** D, where each node is an instruction that finishes in $O(1)$ time or an access (insert/delete/search/update) to the finger structure M, called an M**-call**, that blocks until the result is returned, and each edge represents a dependency due to the parallel programming primitives.

The first design, called $\mathbb{FS}_1$, is a simpler one that processes accesses one batch at a time.

▶ **Theorem 3** ($\mathbb{FS}_1$ Performance). *If P uses $\mathbb{FS}_1$ (as M), then its running time on p processes using any greedy scheduler (i.e. at each step, as many tasks are executed as are available, up to p) is $O\Big(\frac{T_1+F_L}{p} + T_\infty + d \cdot \big((\log p)^2 + \log n\big)\Big)$ for some linearization L of M-calls in D, where T_1 is the number of nodes in D, and T_∞ is the number of nodes on the longest path in D, and d is the maximum number of M-calls on any path in D, and n is the maximum size of M.* [1]

Notice that if M is an ideal concurrent finger structure (i.e. one that takes $O(F_L)$ work), then running P using M on p processors according to the linearization L takes $\Omega(T_{opt})$ worst-case time where $T_{opt} = \frac{T_1+F_L}{p} + T_\infty$. Thus $\mathbb{FS}_1$ gives an essentially optimal time bound except for the "span term" $d \cdot \big((\log p)^2 + \log n\big)$, which adds $O\big((\log p)^2 + \log n\big)$ time per $\mathbb{FS}_1$-call along some path in D.

The second design, called $\mathbb{FS}_2$, uses a complex internal pipeline to reduce the "span term".

▶ **Theorem 4** ($\mathbb{FS}_2$ Performance). *If P uses $\mathbb{FS}_2$, then its running time on p processes using any greedy scheduler is $O\Big(\frac{T_1+F_L}{p} + T_\infty + d \cdot (\log p)^2 + s_L\Big)$ for some linearization L of M-calls in D, where T_1, T_∞, d are defined as in Theorem 3, and s_L is the weighted span of D where each $\mathbb{FS}_2$-call is weighted by its cost according to F_L. Specifically, each access operation on $\mathbb{FS}_2$ with finger distance r according to L is given the weight $\log r + 1$, and s_L is the maximum weight of any path in D. Thus $\mathbb{FS}_2$ gives an essentially optimal time bound up to an extra $O\big((\log p)^2\big)$ time per $\mathbb{FS}_2$-call along some path in D.*

See the full paper for how to extend $\mathbb{FS}_1$ to a general finger structure with f movable fingers, and how to adapt the results for work-stealing schedulers.

Other Related Work

There are many approaches for designing efficient parallel data structures, to make maximal use of parallelism in a multi-processor system, whether with empirical or theoretical efficiency.

For example, Ellen et al. [15] show how to design a non-blocking concurrent binary search tree, with later work analyzing the amortized complexity [14] and generalizing this technique [11]. Another notable concurrent search tree is the CBTree [2, 1], which is based on the splay tree. But despite experimental success, the theoretical access cost for these tree structures may increase with the number of concurrent operations due to contention near the root, and some of them do not even maintain balance (i.e., the height may get large).

Another method is software combining [17, 21, 26], where each process inserts a request into a shared queue and at any time one process is sequentially executing the outstanding requests. This generalizes to parallel combining [6], where outstanding requests are executed in batches on a suitable batch-parallel data structure (similar to implicit batching). These methods were shown to yield empirically efficient concurrent implementations of various common abstract data structures including stacks, queues and priority queues.

In the PRAM model, Paul et al. [27] devised a parallel 2-3 tree where p synchronous processors can perform a sorted batch of p operations on a parallel 2-3 tree of size n in $O(\log n + \log p)$ time. Blelloch et al. [9] show how to increase parallelism of tree operations

[1] To cater to instructions that may not finish in $O(1)$ time (e.g. due to memory contention), it suffices to define T_1 and T_∞ to be the (weighted) work and span (Definition 5) respectively of the program DAG where each M-call is assumed to take $O(1)$ time.

via pipelining. Other similar data structures include parallel treaps [10] and a variety of work-optimal parallel ordered sets [7] supporting unions and intersections with optimal work, but these do not have optimal span. As it turns out, we can in fact have parallel ordered sets with optimal work and span [5, 24].

Nevertheless, the programmer cannot use this kind of parallel data structure as a black box in a high-level parallel program, but must instead carefully coordinate access to it. This difficulty can be eliminated by designing a suitable ***batch-parallel*** data structure and using ***implicit batching*** [3] or ***extended implicit batching*** as presented in [4]. Batch-parallel implementations have been designed for various data structures including weight-balanced B-trees [16], priority queues [6], working-set maps [4] and euler-tour trees [31].

2 Parallel Computation Model

In this section, we describe parallel programming primitives in our model, how a parallel program generates an execution DAG, and how we measure the cost of an execution DAG.

2.1 Parallel Primitives

The parallel finger structures $\mathbb{FS}_1$ and $\mathbb{FS}_2$ in this paper are described and explained as multithreaded data structures that can be used as composable building blocks in a larger parallel program. In this paper we shall focus on the abstract algorithms behind $\mathbb{FS}_1$ and $\mathbb{FS}_2$, relying merely on the following parallel programming primitives (rather than model-specific implementation details, but see the full paper for those):

1. **Threads**: A thread can at any point ***terminate*** itself (i.e. finish running). Or it can ***fork*** another thread, obtaining a pointer to that thread, or ***join*** to a previously forked thread (i.e. wait until that thread terminates). Or it can ***suspend*** itself (i.e. temporarily stop running), after which a thread with a pointer to it can ***resume*** it (i.e. make it continue running from where it left off). Each of these takes $O(1)$ time.
2. **Non-blocking locks**: Attempts to ***acquire*** a non-blocking lock are serialized but do not block. Acquiring the lock succeeds if the lock is not currently held but fails otherwise, and ***releasing*** always succeeds. If k threads concurrently access the lock, then each access finishes within $O(k)$ time.
3. **Dedicated lock**: A dedicated lock is a blocking lock initialized with a constant number of keys, where concurrent threads must use different keys to ***acquire*** it, but ***releasing*** does not require a key. Each attempt to acquire the lock takes $O(1)$ time, and the thread will acquire the lock after at most $O(1)$ subsequent acquisitions of that lock.
4. **Reactivation calls**: A procedure P with no input/output can be encapsulated by a reactivation wrapper, in which it can be run only via ***reactivations***. If there are always at most $O(1)$ concurrent reactivations of P, then whenever a thread ***reactivates*** P, if P is not currently running then it will start running (in another thread forked in $O(1)$ time), otherwise it will run within $O(1)$ time after its current run finishes.

We also make use of basic batch operations, namely filtering, sorted partitioning, joining and merging (see Appendix Appendix A.2), which have easy implementations using arrays in the CREW PRAM model. So $\mathbb{FS}_1$ and $\mathbb{FS}_2$ (using a work-stealing scheduler) can be implemented in the (synchronous) Arbitrary CRCW PRAM model with fetch-and-add, achieving the claimed performance bounds. Actually, $\mathbb{FS}_1$ and $\mathbb{FS}_2$ were also designed to function correctly with the same performance bounds in a much stricter computation model called the QRMW parallel pointer machine model (see Appendix Appendix A.1 for details).

2.2 Execution DAG

The **program DAG** D captures the high-level execution of P, but the actual complete execution of P (including interaction between data structure calls) is captured by the **execution DAG** E (which may be schedule-dependent), in which each node is a basic instruction and the directed edges represent the computation dependencies (such as constrained by forking/joining of threads and acquiring/releasing of blocking locks). At any point during the execution of P, a node in the program/execution DAG is said to be **ready** if its parent nodes have been executed. At any point in the execution, an **active thread** is simply a ready node in E, while a **terminated/suspended thread** is an executed node in E that has no child nodes.

The execution DAG E consists of **program nodes** (specifically P**-nodes**) and **ds (data-structure) nodes**, which are dynamically generated as follows. At the start E has a single program node, corresponding to the start of the program P. Each node could be a **normal instruction** (i.e. basic arithmetic/memory operation) or a **parallel primitive** (see Section 2.1). Each program node could also be a **data structure call**.

When a (ready) node is executed, it may generate child nodes or ***terminate***. A normal instruction generates one child node and no extra edges. A ***join*** generates a child node with an extra edge to it from the ***terminate*** node of the joined thread. A ***resume*** generates an extra child node (the resumed thread) with an edge to it from the ***suspend*** node of the originally suspended thread. Accesses to locks and reactivation calls would each expand to a subDAG comprised of normal instructions and possibly ***fork***/***suspend***/***resume***.

The program nodes correspond to nodes in the program DAG D, and except for data structure calls they generate only program nodes. A call to a data structure M is called an M**-call**. If M is an ordinary (non-batched) data structure, then an M-call generates an M**-node** (and every M-node is a ds node), which thereafter generates only M-nodes except for calls to other data structures (external to M) or returning the result of some operation (generating a program node with an edge to it from the original M-call).

However, if M is an ***(implicitly) batched*** data structure, then all M-calls are automatically passed to the **parallel buffer** for M (see Appendix Appendix A.3). So an M-call generates a **buffer node** corresponding to passing the call to the parallel buffer, as if the parallel buffer for M is itself another data structure and not part of M. Buffer nodes generate only buffer nodes until it notifies M of the buffered M-calls or passes the input batch to M, which generates an M-node. In short, M-nodes exclude all nodes generated as part of the buffer subcomputations (i.e. buffering the M-calls, and notifying M, and flushing the buffer).

2.3 Data Structure Costs

We shall now define work and span of any (terminating) subcomputation of a multithreaded program, i.e. any subset of the nodes in its execution DAG. This allows us to capture the intrinsic costs incurred by a data structure, separate from those of parallel programs using it.

▶ **Definition 5** (Subcomputation Work/Span/Cost). *Take any execution of a parallel program P (on p processors), and take any subset C of nodes in its execution DAG E. The **work** taken by C is the total weight w of C where each node is weighted by the time taken to execute it. The **span** taken by C is the maximum weight s of nodes in C on any (directed) path in E. The **cost** of C is $\frac{w}{p} + s$.*

▶ **Definition 6** (Data Structure Work/Span/Cost). *Take any parallel program P using a data structure M. The **work/span/cost** of M (as used by P) is the work/span/cost of the M-nodes in the execution DAG for P.*

Note that the cost of the entire execution DAG is in fact an upper bound on the actual time taken to run it on a **greedy scheduler**, which on each step assigns as many unassigned ready nodes (i.e. nodes that have been generated but have not been assigned) as possible to available processors (i.e. processors that are not executing any nodes) to be executed.

Moreover, the subcomputation cost is ***subadditive*** across subcomputations. Thus our results are ***composable*** with other algorithms and data structures in this model, since we actually show the following for some linearization L (where F_L, d, n, s_L are as defined in Section 1 Main Results, and N is the total number of calls to the parallel finger structure).

▶ **Theorem 7** ($\mathbb{FS}$ Work/Span Bounds).

- $\mathbb{FS}_1$ *takes* $O(F_L)$ *work and* $O\left(\frac{N}{p} + d \cdot \left((\log p)^2 + \log n\right)\right)$ *span.*
- $\mathbb{FS}_2$ *takes* $O(F_L)$ *work and* $O\left(\frac{N}{p} + d \cdot (\log p)^2 + s_L\right)$ *span.*

Note that the bounds for the work/span of $\mathbb{FS}_1$ and $\mathbb{FS}_2$ are independent of the scheduler. In addition, using any greedy scheduler, the parallel buffer for either finger structure has cost $O\left(\frac{T_1+F_L}{p} + d \cdot \log p\right)$ (Appendix Theorem 13). Therefore our main results (Theorem 3 and Theorem 4) follow from these composable bounds (Theorem 7).

3 Amortized Sequential Finger Structure

In this section we explain an amortized sequential finger structure $\mathbb{FS}_0$ with a fixed finger at each end, which is amenable to parallelization and pipelining due to its ***doubly-exponential segmented structure*** (which was partially inspired by Iacono's working-set structure [22]).

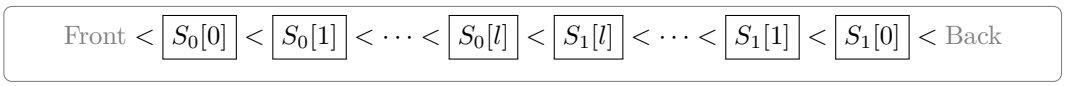

Figure 1 $\mathbb{FS}_0$ Outline; each box $S_i[k]$ represents a 2-3 tree of size $\Theta\left(2^{2^k}\right)$ for $k < l$.

$\mathbb{FS}_0$ keeps the items in order in two halves, the front half stored in a chain of **segments** $S_0[0..l]$, and the back half stored in reverse order in a chain of segments $S_1[0..l]$. Let $c(k) = 2^{2^{k+1}}$ for each $k \in \mathbb{Z}$. Each segment $S_i[k]$ has a **target size** $t(k) = 2 \cdot c(k)$, and a **target capacity** defined to be $[t(k), t(k)]$ if $k < l$ but $[0, t(k)]$ if $k = l$. Each segment stores its items in order in a 2-3 tree. We say that a segment $S_i[k]$ is **balanced** iff its size is within $c(k)$ of its target capacity, and **overfull** iff it has more than $c(k)$ items above target capacity, and **underfull** iff it has more than $c(k)$ items below target capacity. At any time we associate every item x to a unique segment that it **fits** in; x fits in $S_0[k]$ if k is the minimum such that $x \leq \max(S_0[k])$, and that x fits in $S_1[k]$ if k is the minimum such that $x \geq \min(S_1[k])$, and that x fits in $S_0[l]$ if $\max(S_0[l]) < x < \min(S_1[l])$. We shall maintain the invariant that every segment is balanced after each operation is finished.

For each operation on an item x, we find the segment $S_i[k]$ that x fits in, by checking the range of items in $S_0[a]$ and $S_1[a]$ for each a from 0 to l and stopping once k is found, and then perform the desired operation on the 2-3 tree in $S_i[k]$. This takes $O(k + \log(t(k) + c(k))) \subseteq O(2^k) \subseteq O(\log r + 1)$ steps where r is the finger distance of the operation, since $\log_2 r + 1 \geq \log_2 c(k-1) = 2^k$.

After that, if $S_i[k]$ becomes imbalanced, we **rebalance** it by shifting (appropriate) items to or from $S_i[k+1]$ (after creating empty segment $S_i[k+1]$ if it does not exist) to make $S_i[k]$ have target size or as close as possible (via a suitable split then join of the 2-3 trees), and

then $S_i[k+1]$ is removed if it is the last segment and is now empty. After the rebalancing, $S_i[k]$ will not only be balanced but also have size within its target capacity. But now $S_i[k+1]$ may become imbalanced, so the rebalancing may cascade.

Finally, if one chain $S_i[0..l']$ is longer than the other chain $S_j[0..l]$, it must be that $l' = l+1$, so we **rebalance** the chains as follows: If $S_j[l]$ is below target size, shift items from $S_i[l']$ to $S_j[l]$ to fill it up to target size. If $S_j[l]$ is (still) below target size, remove the now empty $S_i[l']$, otherwise add a new empty segment $S_j[l+1]$.

Each rebalancing cascade may take $\Theta(\log n)$ steps, but the total rebalancing cost is only $O(1)$ amortized steps per operation, which we can prove via an accounting argument: We are given 1 credit for each operation, and use it to maintain a ***credit invariant*** that each segment $S_i[k]$ with q items beyond (i.e. above or below) its target capacity has at least $q \cdot 2^{-k}$ stored credits, and use the stored credits to pay for all rebalancing. Whenever a segment $S_i[k]$ is rebalanced, it must have had q items beyond its target capacity for some $q > c(k)$, and so had at least $q \cdot 2^{-k}$ stored credits. Also, the rebalancing itself takes $O(\log(t(k)+q) + \log(t(k+1)+c(k+1)+q)) \subseteq O(\log q) \subseteq O\big(q \cdot 2^{-k}\big)$ steps, after which $S_i[k+1]$ needs at most $q \cdot 2^{-(k+1)}$ extra stored credits. Thus the stored credits at $S_i[k]$ can be used to pay for both the rebalancing and any extra stored credits needed by $S_i[k+1]$. Whenever the chains are rebalanced, it can be paid for by the last segment rebalancing (which created or removed a segment), and no extra stored credits are needed.

4 Simpler Parallel Finger Structure

We now present our simpler parallel finger structure $\mathbb{FS}_1$. The idea is to use the amortized sequential finger structure $\mathbb{FS}_0$ (Section 3) and execute operations in batches. We group each pair of segments $S_0[k]$ and $S_1[k]$ into one **section** $S[k]$, and we say that an item x **fits in** the sections $S[j..k]$ iff x fits in some segment in $S[j..k]$.

Each segment is stored in an **optimal batch-parallel map** [24, 8], which supports:

- **Unsorted batch search:** Search for an unsorted batch of b items, tagging each search with the result, within $O(b \cdot \log n)$ work and $O(\log b \cdot \log n)$ span, where n is the map size.
- **Sorted batch access:** Perform an item-sorted batch of b operations on distinct items, tagging each operation with the result, within $O(b \cdot \log n)$ work and $O(\log b + \log n)$ span, where n is the map size before the batch access.
- **Split**: Split a map M of size k around a pivot rank r into maps M_1, M_2 where M_1 contains the first r items in M, and M_2 contains the last $k-r$ items in M, within $O(\log k)$ work/span.
- **Join**: Join maps M_1, M_2 of total size k where the greatest item in M_1 is less than the least item in M_2, within $O(\log k)$ work/span.

For each section $S[k]$, we can perform a batch of b operations on it within $O(b \cdot \log c(k))$ work and $O(\log b + \log c(k))$ span if we have the batch sorted. Excluding sorting, the total work would satisfy the finger bound just like in $\mathbb{FS}_0$. But we cannot afford to sort the input batch right at the start, because if the batch had b searches of distinct items all with finger distance $O(1)$, then it would take $\Omega(b \cdot \log b)$ work and exceed our finger bound budget of $O(b)$.

We can solve this by splitting the sections into two slabs, where the first slab comprises the first $\log\log(2b)$ sections, and passing the batch through a preliminary phase in which we merely perform an unsorted search of the relevant items in the first slab, and eliminate operations on items that fit in the first slab but are neither found nor to be inserted.

This preliminary phase takes $O(\log c(k))$ work per operation and $O(\log b \cdot \log c(k))$ span at each section $S[k]$. We then sort the uneliminated operations and execute them on the appropriate slab. For this, ordinary sorting still takes too much work as there can be many

operations on the same item, but it turns out that the finger bound budget is enough to pay for entropy-sorting (Appendix Definition 15), which takes $O\left(\log \frac{b}{q} + 1\right)$ work for each item that occurs q times in the batch. Rebalancing the segments and chains is a little tricky, but if done correctly it takes $O(1)$ amortized work per operation. Therefore we achieve work-optimality while being able to process each batch within $O\left((\log b)^2 + \log n\right)$ span. The details are below.

4.1 Description of $\mathbb{FS}_1$

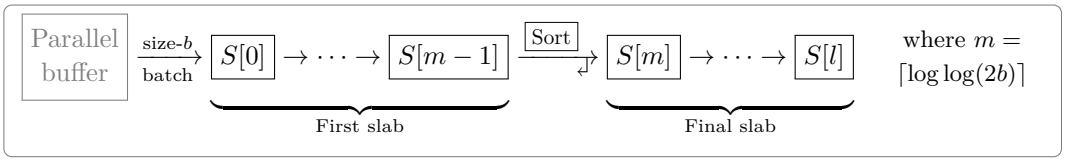

Figure 2 $\mathbb{FS}_1$ Outline; each batch is sorted only after being filtered through the smaller sections.

$\mathbb{FS}_1$-calls are put into the parallel buffer (Section 2) for $\mathbb{FS}_1$. Whenever the previous batch is done, $\mathbb{FS}_1$ flushes the parallel buffer to obtain the next batch B. Let b be the size of B, and we can assume $b > 1$. Based on b, the sections in $\mathbb{FS}_1$ are conceptually divided into two slabs, the **first slab** comprising sections $S[0..m-1]$ and the **final slab** comprising sections $S[m..l]$, where $m = \lceil \log \log(2b) \rceil + 1$ (where log is the binary logarithm). The items in each segment are stored in a batch-parallel map.

$\mathbb{FS}_1$ processes the input batch B in four phases:

1. **Preliminary phase**: For each first slab section $S[k]$ in order (i.e. k from 0 to $m-1$) do as follows:
   a. Perform an unsorted search in each segment in $S[k]$ for all the items relevant to the remaining batch B' (of direct pointers into B), and tag the operations in the original batch B with the results.
   b. Remove all operations on items that fit in $S[k]$ from the remaining batch B'.
   c. Skip the rest of the first slab if B' becomes empty.
2. **Separation phase**: Partition B based on the tags into three parts and handle each part separately as follows:
   a. ***Ineffectual operations*** (on items that fit in the first slab but are neither found nor to be inserted): Return the results.
   b. ***Effectual operations*** (on items found in or to be inserted into the first slab): Entropy-sort (Appendix Definition 15) them in order of access type (search, update, insertion, deletion) with deletions last, followed by item, combining operations of the same access type on the same item into one **group-operation** that is treated as a single operation whose **effect** is the last operation in that group. Each group-operation is stored in a leaf-based binary tree with height $O(\log b)$ (but not necessarily balanced), and the combining is done during the entropy-sorting itself.
   c. ***Residual operations*** (on items that do not fit in the first slab): Sort them while combining operations in the same manner as for effectual operations.

3. **Execution phase**: Execute the effectual operations as a batch on the first slab, and then execute the residual operations as a batch on the final slab, for each slab doing the following at each section $S[k]$ in order (small to big):
   a. Let $G_{1..4}$ be the partition of the batch of operations into the 4 access types (deletions last), each G_a sorted by item.
   b. For each segment $S_i[k]$ in $S[k]$, and for each a from 1 to 4, cut out the operations that fit in $S_i[k]$ from G_a, and perform those operations (as a sorted batch) on $S_i[k]$, and then return their results.
   c. Skip the rest of the slab if the batch becomes empty.
4. **Rebalancing phase**: Rebalance all the segments and chains, by doing the following:
   a. ***Segment rebalancing***: For each chain S_i, for each segment $S_i[k]$ in S_i in order (small to big):
      i. If $k > 0$ and $S_i[k-1]$ is overfull, make $S_i[k-1]$ have target size by shifting items from it to $S_i[k]$.
      ii. If $k > 0$ and $S_i[k-1]$ is underfull and $S_i[k]$ has at least $\frac{c(k)}{2}$ items, let $S_i[k']$ be the first underfull segment in S_i, and **fill** $S_i[k'..k-1]$ using $S_i[k]$ as follows: for each j from $k-1$ down to k', shift items from $S_i[j+1]$ to $S_i[j]$ to make $S_i[k'..j]$ have total size $\sum_{a=k'}^{j} t(a)$ or as close as possible, and then remove $S_i[j+1]$ if it is emptied.
      iii. If $S_i[k]$ is (still) overfull and is the last segment in S_i, create a new (empty) segment $S_i[k+1]$.
      iv. Skip the rest of the current slab if $S_i[k]$ is balanced and the execution phase had skipped $S[k]$.
   b. ***Chain rebalancing***: After that, if one chain S_i is longer than the other chain S_j, repeat the following until the chains are the same length:
      i. Let the current chains be $S_i[0..k]$ and $S_j[0..k']$. Create new (empty) segments $S_j[k'+1..k]$, and shift all items from $S_i[k]$ to $S_j[k]$, and then **fill** the underfull segments in $S_j[k'..k-1]$ using $S_j[k]$ (as in step 4aii). If $S_j[k]$ is (now) empty again, remove $S[k]$.

4.2 Analysis of $\mathbb{FS}_1$

It is not hard to prove that every segment is balanced (just) after the rebalancing phase. (See the full paper for the details.) Based on that, we shall now bound the work done by $\mathbb{FS}_1$.

▶ **Definition 8** (Inward Order). *Take any sequence A of map operations and let I be the set of items accessed by operations in A. Define the **inward distance** of an operation in A on an item x to be* $\min(\text{size}(I_{\leq x}), \text{size}(I_{\geq x}))$. *We say that A is in **inward order** iff its operations are in order of (non-strict) increasing inward distance. Naturally, we say that A is in **outward order** iff its reverse is in inward order.*

▶ **Theorem 9** ($\mathbb{FS}_1$ Work). *$\mathbb{FS}_1$ takes $O(F_L)$ work for some linearization L of $\mathbb{FS}_1$-calls in D.*

Proof. Let L^* be a linearization of $\mathbb{FS}_1$-calls in D such that:

- Operations on $\mathbb{FS}_1$ in earlier input batches are before those in later input batches.
- The operations within each batch are ordered as follows:
  1. Ineffectual operations are before effectual/residual operations.
  2. Effectual/residual operations are in order of access type (deletions last).
  3. Effectual insertions are in inward order, and effectual deletions are in outward order.
  4. Operations in each group-operation are consecutive and in the same order as in that group.

Let L' be the same as L^* except that in point 3 effectual deletions are ordered so that those on items in earlier sections are later (instead of outward order). Now consider each input batch B of b operations on $\mathbb{FS}_1$.

In the preliminary and execution phases, each section $S[a]$ takes $O(2^a)$ work per operation. Thus each operation in B with finger distance r according to L' on an item x that was found to fit in section $S[k]$ takes $O\left(\sum_{a=0}^{k} 2^a\right) = O\left(2^k\right) \subseteq O(\log r + 1)$ work, because $r \geq \sum_{a=0}^{k-1} c(a) + 1 \geq \frac{1}{2}c(k-1)$ if $S[k]$ is in the first slab (since earlier effectual operations in B did not delete items in $S[0..k-1]$), and $r \geq \sum_{a=0}^{k-1} c(a) - b \geq \frac{1}{2}c(k-1)$ if $S[k]$ is in the final slab (since $b \leq \frac{1}{2}c(m-1)$). Therefore these phases take $O(F_{L'})$ work in total.

Let G be the effectual operations in B as a subsequence of L^*. Entropy-sorting G takes $O(H+b)$ work (Appendix Theorem 16), where H is the entropy of G (i.e. $H = \sum_{i=1}^{b} \log \frac{b}{q_i}$ where q_i is the number of occurrences of the i-th operation in G). Partition G into 3 parts: searches/updates G_1 and insertions G_2 and deletions G_3. And let H_j be the entropy of G_j. Then $H = \sum_{j=1}^{3} H_j + \sum_{i=1}^{b} \log \frac{b}{b_i}$ where b_i is the number of operations in the same part of G as the i-th operation in G, and $\sum_{i=1}^{b} \log \frac{b}{b_i} \leq b \cdot \log\left(\frac{1}{b}\sum_{i=1}^{b} \frac{b}{b_i}\right) = b \cdot \log 3$ by Jensen's inequality. Thus entropy-sorting G takes $O\left(\sum_{j=1}^{3} H_j + b\right)$ work. Let C_j be the cost of G_j according to F_{L^*}. Since each operation in G_j has inward distance (with respect to G_j) at most its finger distance according to L^*, we have $H_j \in O(C_j)$ (Appendix Theorem 14), and hence entropy-sorting takes $O(F_{L^*})$ work in total.

Sorting the residual operations in B (that do not fit in the first slab) takes $O(\log b) \subseteq O(\log r)$ work per operation with finger distance r according to L^*, since $r \geq c(m-1) \geq 2b$.

Therefore the separation phase takes $O(F_{L^*})$ work in total. Finally, the rebalancing phase takes $O(1)$ amortized work per operation, as we shall prove in the next lemma. Thus $\mathbb{FS}_1$ takes $O(\max(F_{L^*}, F_{L'}))$ total work. ◀

▶ **Lemma 10** ($\mathbb{FS}_1$ Rebalancing Work). *The rebalancing phase of $\mathbb{FS}_1$ takes $O(1)$ amortized work per operation.*

Proof. We shall maintain the credit invariant that each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits. The execution phase clearly increases the total stored credits needed by at most 1 per operation, which we can pay for. We now show that the invariant can be preserved after the segment rebalancing and the chain rebalancing.

During the segment rebalancing (step 4a), each shift is performed between some neighbouring segments $S_i[k]$ and $S_i[k+1]$, where $S_i[k]$ has $t(k)+q$ items and $S_i[k+1]$ has $t(k+1)+q'$ items just before the shift, and $|q| > c(k)$. The shift clearly takes $O(\log(t(k)+q)+\log(t(k+1)+q'))$ work. If $q' < 2 \cdot t(k+1)$ then this is obviously just $O(\log t(k) + \log|q|)$ work. But if $q' > 2 \cdot t(k+1)$, then $S_i[k+1]$ will also be rebalanced in step 4ai of the next segment balancing iteration, since at most $\sum_{a=0}^{k} t(a) \leq t(k+1)$ items will be shifted from $S_i[k+1]$ to $S_i[k]$ in step 4aii, and hence $S_i[k+1]$ will still have at least q' items. In that case, the second term $O(\log(t(k+1)+q')))$ in the work bound for this shift can be bounded by the first term of the work bound for the subsequent shift from $S_i[k+1]$ to $S_i[k+2]$, since $\log(t(k+1)+q') \in O(\log q')$. Therefore in any case we can treat this shift as taking only $O(\log t(k) + \log|q|) \subseteq O(\log|q|) \subseteq O\left(|q| \cdot 2^{-k}\right)$ work.

Now consider the two kinds of segment rebalancing:

- *Overflow*: step 4ai shifts items from overfull $S_i[k]$ to $S_i[k+1]$, where $S_i[k]$ has $t(k)+u$ items just before the shift. After the shift, $S_i[k]$ has target size and needs no stored credits, and $S_i[k+1]$ would need at most $u \cdot 2^{-(k+1)}$ extra stored credits. Thus the $u \cdot 2^{-k}$ credits stored at $S_i[k]$ can pay for both the shift and the needed extra stored credits.

- *Fill*: step 4aii fills some underfull segments $S_i[k'..k]$ using $S_i[k+1]$, where $S_i[j]$ has $t(j) - u_i(j)$ items just before the fill, for each $j \in [k'..k]$. After the fill, every segment in $S_i[k'..k]$ would have target size and need no stored credits, and $S_i[k+1]$ will need at most $\left(\sum_{j=k'}^{k} u_i(j)\right) \cdot 2^{-(k+1)} \leq \frac{1}{2} \sum_{j=k'}^{k} \left(u_i(j) \cdot 2^{-j}\right)$ extra stored credits, which can be paid for by using half the credits stored at each segment in $S_i[k'..k]$. The other half of the $u_i(j) \cdot 2^{-j}$ credits stored at $S_i[j]$ suffices to pay for the shift from $S_i[j+1]$ to $S_i[j]$ for each $j \in [k'..k]$.

If chain rebalancing (step 4b) is performed, segment rebalancing must have created or removed some segment, in which case there were enough deletions or shifted items that the work done by chain rebalancing can be ignored. The details are in the full paper. ◀

The span bound for $\mathbb{FS}_1$ is also relegated to the full paper.

5 Faster Parallel Finger Structure

Although $\mathbb{FS}_1$ has optimal work and a small span, it is possible to reduce the span even further, intuitively by pipelining the batches in some fashion so that an expensive access in a batch does not hold up the next batch.

As with $\mathbb{FS}_1$, we need to split the sections into two slabs, but this time we fix the first slab at m sections where $m \in \log \Theta(\log p)$ so that we can pipeline just the final slab. We need to allow big enough batches so that operations that are delayed because earlier batches are full can count their delay against the total work divided by p. But to keep the span of the sorting phase down to $O\big((\log p)^2\big)$, we need to restrict the batch size. It turns out that restricting to batches of size at most p^2 works.

We cannot pipeline the first slab (particularly the rebalancing), but the preliminary phase and separation phase would only take $O\big((\log p)^2\big)$ span. The execution phase and rebalancing phases are still carried out as before on the first slab, taking $O\big((\log p)^2\big)$ span, but execution and rebalancing on the final slab are pipelined, by having each final slab section $S[k]$ process the batch passed to it and rebalance the preceding segments $S_0[k-1]$ and $S_1[k-1]$ if necessary.

One ***key challenge*** is how to guarantee that such local rebalancing in the final slab is always possible and always sufficient. To ensure that, we do not allow $S[k]$ to proceed if it is imbalanced or if there are more than $c(k)$ pending operations in the buffer to $S[k+1]$. In such a situation, $S[k]$ must stop and reactivate $S[k+1]$, which would clear its buffer and rebalance $S[k]$ before restarting $S[k]$. It may be that $S[k+1]$ also cannot proceed for the same reason and is stopped in the same manner, and so $S[k]$ may be delayed by such a stop for a long time. But by a suitable accounting argument we can bound the total delay due to all such stops by the total work divided by p. Similarly, we do not allow the first slab to run (on a new batch) if $S[m-1]$ is imbalanced or there are more than $c(m-1)$ pending operations in the buffer to $S[m]$.

Finally, we use an odd-even locking scheme to ensure that the segments in the final slab do not interfere with each other yet can proceed at a consistent pace. The details are below.

5.1 Description of $\mathbb{FS}_2$

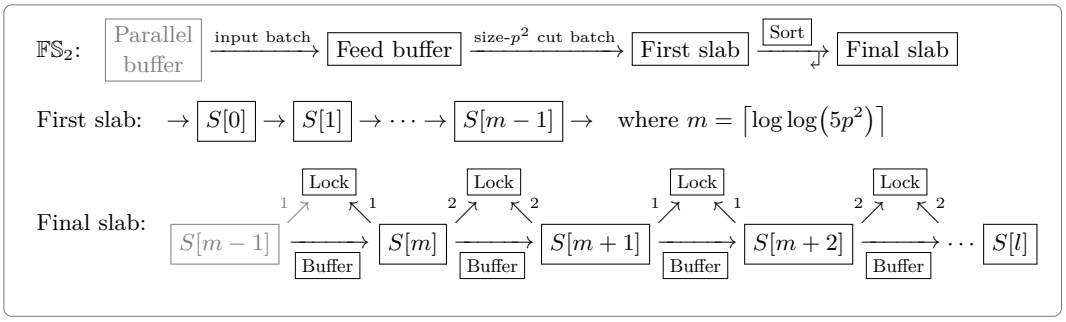

Figure 3 $\mathbb{FS}_2$ Sketch; the final slab is pipelined, facilitated by locks between adjacent sections.

We will need the **bunch** structure (Appendix Definition 12) for aggregating batches, which is an unsorted set supporting both addition of a batch of new elements within $O(1)$ work/span and conversion to a batch within $O(b)$ work and $O(\log b)$ span if it has size b.

$\mathbb{FS}_2$ has the same sections as in $\mathbb{FS}_1$, with the **first slab** comprising the first $m = \lceil \log\log(5p^2) \rceil$ sections, and the **final slab** comprising the other sections. $\mathbb{FS}_2$ uses a **feed buffer**, which is a queue of bunches each of size p^2 except the last (which can be empty). Whenever $\mathbb{FS}_2$ is notified of input (by the parallel buffer), it reactivates the first slab.

Each section $S[k]$ in the final slab has a **buffer** before it (for pending operations from $S[k-1]$), which for each access type uses an optimal batch-parallel map to store bunches of group-operations of that type, where operations on the same item are in the same bunch. When a batch of group-operations on an item is inserted into the buffer, it is simply added to the correct bunch. Whenever we count operations in the buffer, we shall count them individually even if they are on the same item. The first slab and each final slab section also has a **deferred flag**, which indicates whether its run is deferred until the next section has run. Between every pair of consecutive sections starting from after $S[m-1]$ is a **neighbour-lock**, which is a dedicated lock (see Section 2.1) with 1 key for each arrow to it in Figure 3.

Whenever the first slab is reactivated, it runs as follows:

1. If the parallel buffer and feed buffer are both empty, terminate.
2. Acquire the neighbour-lock between $S[m-1]$ and $S[m]$. (Skip steps 2 to 4 and steps 8 to 10 if $S[m]$ does not exist.)
3. If $S[m-1]$ has any imbalanced segment or $S[m]$ has more than $c(m-1)$ operations in its buffer, set the first slab's deferred flag and release the neighbour-lock, and then reactivate $S[m]$ and terminate.
4. Release the neighbour-lock.
5. Let q be the size of the last bunch F in the feed buffer. Flush the parallel buffer (if it is non-empty) and cut the input batch of size b into small batches of size p^2 except possibly the first and last, where the first has size $\min(b, p^2 - q)$. Add that first small batch to F, and append the rest as bunches to the feed buffer.
6. Remove the first bunch from the feed buffer and convert it into a batch B, which we call a **cut batch**.

7. Process B using the same four phases as in $\mathbb{FS}_1$ (Section 4.1), but restricted to the first slab (i.e. execute only the effectual operations on the first slab, and do segment rebalancing only on the first slab, and do chain rebalancing only if $S[m]$ had not existed before this processing). Furthermore, do not update $S[m-1]$'s segments' sizes until after this processing (so that section $S[m]$ in step 4 will not find any of $S[m-1]$'s segments imbalanced until the first slab rebalancing phase has finished).
8. Acquire the neighbour-lock between $S[m-1]$ and $S[m]$.
9. Insert the residual group-operations (on items that do not fit in the first slab) into the buffer of $S[m]$, and then reactivate $S[m]$.
10. Release the neighbour-lock.
11. Reactivate itself.

Whenever a final slab section $S[k]$ is reactivated, it runs as follows:

1. Acquire the neighbour-locks (between $S[k]$ and its neighbours) in the order given by the arrow number in Figure 3.
2. If $S[k]$ has any imbalanced segment or $S[k+1]$ (exists and) has more than $c(k)$ operations in its buffer, set $S[k]$'s deferred flag and release the neighbour-locks, and then reactivate $S[k+1]$ and terminate.
3. For each access type, flush and process the (sorted) batch G of bunches of group-operations of that type in its buffer as follows:
   a. Convert each bunch in G to a batch of group-operations.
   b. For each segment $S_i[k]$ in $S[k]$, cut out the group-operations on items that fit in $S_i[k]$ from G, and perform them (as a sorted batch) on $S_i[k]$, and then fork to return the results of the operations (according to the order within each group-operation).
   c. If G is non-empty (i.e. has leftover group-operations), insert G into the buffer of $S[k+1]$ and then reactivate $S[k+1]$.
4. Rebalance locally as follows (essentially like in $\mathbb{FS}_1$):
   a. For each segment $S_i[k]$ in $S[k]$:
      i. If $S_i[k-1]$ is overfull, shift items from $S_i[k-1]$ to $S_i[k]$ to make $S_i[k-1]$ have target size.
      ii. If $S_i[k-1]$ is underfull, shift items from $S_i[k]$ to $S_i[k-1]$ to make $S_i[k-1]$ have target size, and then remove $S_i[k]$ if it is emptied.
      iii. If $S_i[k]$ is (still) overfull and is the last segment in S_i, create a new segment $S_i[k+1]$ and reactivate it.
   b. If $S[k]$ is (still) the last section, but chain S_i is longer than chain S_j:
      i. Create a new segment $S_j[k]$ and shift all items from $S_i[k]$ to $S_j[k]$.
      ii. If $S_j[k-1]$ is (now) underfull, shift items from $S_j[k]$ to $S_j[k-1]$ to make $S_j[k-1]$ have target size.
      iii. If $S_j[k]$ is (now) empty again, remove $S[k]$.
5. If $k = m$, and the first slab is deferred, clear its deferred flag then reactivate it.
6. If $k > m$, and $S[k-1]$ is deferred, clear its defered flag then reactivate it.
7. Release the neighbour-locks.

5.2 Analysis of $\mathbb{FS}_2$

See the full paper for the proofs. The first step is to establish the $\mathbb{FS}_2$ Balance Invariants:

▶ **Lemma 11** ($\mathbb{FS}_2$ Balance Invariants). *$\mathbb{FS}_2$ satisfies the following invariants:*

1. *When the first slab is not running, every segment in $S_i[0..m-2]$ is balanced and $S_i[m-1]$ has at most $2 \cdot t(m-1)$ items.*
2. *When a final slab section $S[k]$ rebalances a segment in $S[k-1]$ (in step 4a), it will make that segment have size $t(k-1)$.*
3. *Just after the last section finishes running without creating new sections, the segments in $S[k]$ are balanced and both chains have the same length.*
4. *Each final slab section $S[k]$ always has at most $2 \cdot c(k-1)$ operations in its buffer.*
5. *Each final slab segment $S_i[k]$ always has at most $2 \cdot t(k)$ items, and at least $c(k-1)$ items unless $S[k]$ is the last section.*

Using these invariants, we can prove the work bound for $\mathbb{FS}_2$ (as stated in Theorem 7), by linearizing operations that finish during the first slab run or final slab section according to when that run finishes, and linearizing operations that finish in the same first slab run in the same way as in the proof for $\mathbb{FS}_1$ (see Theorem 9). This relies on a supporting lemma that all rebalancing done by $\mathbb{FS}_2$ takes $O(1)$ amortized work per operation, which can be proven by a credit invariant like the one used for $\mathbb{FS}_1$ (see Lemma 10): Each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits, and each unfinished operation carries 1 credit with it.

The span bound for $\mathbb{FS}_2$ (as stated in Theorem 7) requires another credit invariant: For $k \geq m-1$, each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits, and each operation in $S[k+1]$'s buffer carries 2^{-k} credits with it. This invariant is used to bound the deferment delay (the delay that an operation may face due to deferred section runs), where we use the credits to pay for p times the deferment delay, to show that the deferment delay is at most $O\left(\frac{1}{p}\right)$ per operation on $\mathbb{FS}_2$. The rest of the delay incurred by each operation can be bounded by tracing its path through the sections and using the fact that the neighbour-locking scheme ensures that the first slab contributes $O\big((\log p)^2\big)$ delay and each final slab section $S[k]$ contributes $O\big(2^k\big)$ delay.

References

1 Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing*, 27(6):393–417, 2014.

2 Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2012.

3 Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 84–95. ACM, 2014.

4 Kunal Agrawal, Seth Gilbert, and Wei Quan Lim. Parallel Working-Set Search Structures. In *Proceedings of the 30th ACM symposium on Parallelism in algorithms and architectures*, pages 321–332. ACM, 2018. `arXiv:1805.05787`.

5 Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 291–300. IEEE, 2016.

6 Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel Combining: Benefits of Explicit Synchronization. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2018.11.

7 Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2016.

8 Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal Parallel Algorithms in the Binary-Forking Model. *arXiv preprint*, 2019. arXiv:1903.04650.

9 Guy E. Blelloch and Margaret Reid-Miller. Pipelining with Futures. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 249–259, New York, NY, USA, 1997. ACM. doi:10.1145/258492.258517.

10 Guy E. Blelloch and Margaret Reid-Miller. Fast Set Operations Using Treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998. doi:10.1145/277651.277660.

11 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, volume 49, pages 329–342. ACM, 2014.

12 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

13 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

14 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 332–340. ACM, 2014.

15 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM. doi:10.1145/1835698.1835736.

16 Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *International Symposium on Experimental Algorithms*, pages 111–122. Springer, 2014.

17 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *PPoPP*, pages 257–266, 2012. doi:10.1145/2145816.2145849.

18 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

19 Michael T Goodrich and S Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *Journal of the ACM (JACM)*, 43(2):331–361, 1996.

20 Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

21 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010. doi:10.1145/1810479.1810540.

22 John Iacono. Alternatives to splay trees with O(log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

23 Intel Corporation. *Intel Cilk Plus Language Extension Specification, Version 1.1*, 2013. Document 324396-002US. Available from `http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm`.

24 Wei Quan Lim. Optimal Multithreaded Batch-Parallel 2-3 Trees. *arXiv*, 2019. `arXiv:1905.05254`.

25 OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. Available from `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

26 Y. Oyama, K. Taura, and A. Yonezawa. Executing Parallel Programs With Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, 1999.

27 Wolfgang Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries on 2–3 trees. *Automata, Languages and Programming*, pages 597–609, 1983.

28 James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

29 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

30 The Task Parallel Library. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`, October 2007. URL: `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`.

31 Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-Parallel Euler Tour Trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

A Appendix

Here we spell out the model details, building blocks and supporting theorems used in our paper. More details and proofs can be found in the full paper.

A.1 QRMW Pointer Machine Model

QRMW stands for **queued read-modify-write**, as described in [13]. In this contention model, asynchronous processors perform memory accesses via read-modify-write (RMW) operations (including read, write, test-and-set, fetch-and-add, compare-and-swap), which are supported by almost all modern architectures. Also, to capture contention costs, multiple memory requests to the same memory cell are FIFO-queued and serviced one at a time, and the processor making each memory request is blocked until the request has been serviced.

The QRMW pointer machine model, introduced in [4], extends the parallel pointer machine model in [19] to RMW operations. An RMW operation can be performed on any memory cell via a pointer to the memory node that it belongs to. All operations except for memory accesses take one step each. Accesses to each memory cell are FIFO-queued to be serviced, and the first access in the queue (if any) is serviced at each time step. The processor making each memory request is blocked until the request has been serviced.

A.2 Parallel Batch Operations

We rely on the following basic operations on batches:

- Split a given batch of n items into left and right parts around a given position, within $O(\log n)$ work/span.
- Partition a given batch of n items into lower and upper parts around a given pivot, within $O(n)$ work and $O(\log n)$ span.
- Partition a sorted batch of n items around a sorted batch of k pivots, within $O(k \cdot \log n)$ work and $O(\log n + \log k)$ span.

- Join a batch of batches with n total items, within $O(n)$ work and $O(\log n)$ span.
- Merge two sorted batches with n total items, optionally combining duplicates, within $O(n)$ work and $O(\log n)$ span if the combining procedure takes $O(1)$ work/span.

These can be implemented in the QRMW pointer machine model [24] with each batch stored as a **BBT** (leaf-based balanced binary tree). They can also be implemented (more easily) in the binary forking model in [8] with each batch stored in an array.

We also rely on the bunch data structure, defined as follows.

▶ **Definition 12** (Bunch Structure). *A **bunch** is an unsorted set supporting addition of any batch of new elements within $O(1)$ work/span and conversion to a batch within $O(b)$ work and $O(\log b)$ span if it has size b. A bunch can be implemented using a complete binary tree with batches at the leaves, with a linked list threaded through each level to support adding a new batch as a leaf in $O(1)$ work/span. To convert a bunch to a batch, we treat the bunch as a batch of batches and parallel join all the batches.*

A.3 Parallel Buffer

To facilitate extended implicit batching, we can use any parallel buffer implementation that takes $O(p + b)$ work and $O(\log p + \log b)$ span per batch of size b (on p processors), as long as any operation that arrives is (regardless of the scheduler) within $O(1)$ span included in the batch that is being flushed or in the next batch, and there are always at most $\frac{1}{2}p + q$ ready buffer nodes (active threads of the buffer) where q is the number of operations that are currently buffered or being flushed. This would entail the following parallel buffer overhead [4].

▶ **Theorem 13** (Parallel Buffer Cost). *Take any program P using an implicitly batched data structure M, run using any greedy scheduler. Then the cost (Definition 6) of the parallel buffer for M is $O\left(\frac{T_1+w}{p} + d \cdot \log p\right)$, where T_1 is the work of the P-nodes, and w is the work taken by M, and d is the maximum number of M-calls on any path in the program DAG D.*

▶ Remark. In general, if a program uses a fixed number of implicitly batched data structures, then running it using a greedy scheduler takes $O\left(\frac{T_1+w^*}{p} + T_\infty + s^* + d^* \cdot \log p\right)$ time, where w^* is the total work of all the data structures, and s^* is the total span of all the data structures, and d^* is the maximum number of data structure calls on any path in the program DAG.

The parallel buffer for each data structure M can be implemented using a static BBT, with a sub-buffer at each leaf node, one per processor, and a flag at each internal node. Each sub-buffer stores its operations as the leaves of a complete binary tree with a linked list through each level. Whenever a thread τ makes a call to M, the processor running τ suspends it and inserts the call together with a callback (i.e. a structure with a pointer to τ and a field for the result) into the sub-buffer for that processor. Then the processor walks up the BBT from leaf to root, test-and-setting each flag along the way, terminating if it was already set. On reaching the root, the processor notifies M (by reactivating it). M can eventually return the result of the call via the callback (i.e. by updating the result field and then resuming τ).

Whenever the buffer is flushed (by M), all sub-buffers are swapped out by a parallel recursion on the BBT, replaced by new sub-buffers in a newly constructed static BBT. We then wait for all pending insertions into the old sub-buffers to be completed, before joining their contents into a single batch to be returned (to M). To do so, each processor i has a flag y_i initialized to *true*, and a thread field ϕ_i initialized to *null*. Whenever it inserts an

M-call X, it sets $y_i := false$, then inserts X into the (current) sub-buffer, then resumes ϕ_i if TestAndSet$(y_i) = true$. To wait for pending insertions into the old sub-buffer for processor i, we store a pointer to the current thread in ϕ_i and then suspend it if TestAndSet$(y_i) = false$.

A.4 Sorting Theorems

Let S be the set of possible items, linearly ordered by a given comparison function.

▶ **Theorem 14** (Maximum Finger Bound). *Take any sequence I in S^n with in-order item frequencies $q_{1..u}$, namely the i-th smallest item in I (not counting duplicates) occurs q_i times in I. Then the* ***maximum finger bound*** *for I, defined as $MF_I = \sum_{i=1}^{u} q_i \cdot (\log \min(i, u + 1 - i) + 1)$, satisfies $MF_I \in \Omega(H)$ where $H = \sum_{i=1}^{u} \left(q_i \cdot \log \frac{n}{q_i} \right)$.*

▶ **Definition 15** (Parallel Entropy-Sort). *Define a* ***bundle*** *of an item x to be a BT (binary tree) in which every leaf has a tagged copy of x. Let $PESort$ be the parallel merge-sort variant that does the following on an input batch I of items (I has subtrees $I.left$ and $I.right$):*

> *If $I.size \leq 1$, return I. Otherwise, compute $A = PEMerge(I.left)$ and $B = PEMerge(I.right)$ in parallel, and then parallel merge (Appendix A.2) A and B into an item-sorted batch C of bundles, combining bundles of the same item into one by simply making them the child subtrees of a new bundle, and then return C.*

Then $PESort(I)$ returns an item-sorted batch of bundles, with one bundle (of all the tagged copies) for each distinct item in I, and clearly each bundle has height at most $I.height$.

▶ **Theorem 16** ($PESort$ Costs). *$PESort$ sorts every sequence in S^n with item frequencies $q_{1..u}$ within $O(H + n)$ work and $O\big((\log n)^2\big)$ span, where $H = \sum_{i=1}^{u} \left(q_i \cdot \ln \frac{n}{q_i} \right)$.*