Optimal Parallel Algorithms in the Binary-Forking Model

Guy E. Blelloch
guyb@cs.cmu.edu
Carnegie Mellon University

Jeremy T. Fineman
jfineman@cs.georgetown.edu
Georgetown University

Yan Gu
ygu@cs.ucr.edu
University of California,
Riverside

Yihan Sun
yihans@cs.ucr.edu
University of California,
Riverside

ABSTRACT

In this paper we develop optimal algorithms in the binary-forking model for a variety of fundamental problems, including sorting, semisorting, list ranking, tree contraction, range minima, and ordered set union, intersection and difference. In the binary-forking model, tasks can only fork into two child tasks, but can do so recursively and asynchronously. The tasks share memory, supporting reads, writes and test-and-sets. Costs are measured in terms of work (total number of instructions), and span (longest dependence chain).

The binary-forking model is meant to capture both algorithm performance and algorithm-design considerations on many existing multithreaded languages, which are also asynchronous and rely on binary forks either explicitly or under the covers. In contrast to the widely studied PRAM model, it does not assume arbitrary-way forks nor synchronous operations, both of which are hard to implement in modern hardware. While optimal PRAM algorithms are known for the problems studied herein, it turns out that arbitrary-way forking and strict synchronization are powerful, if unrealistic, capabilities. Natural simulations of these PRAM algorithms in the binary-forking model (i.e., implementations in existing parallel languages) incur an $\Omega(\log n)$ overhead in span. This paper explores techniques for designing optimal algorithms when limited to binary forking and assuming asynchrony. All algorithms described in this paper are the first algorithms with optimal work and span in the binary-forking model. Most of the algorithms are simple. Many are randomized.

1 INTRODUCTION

In this paper we present several results on the binary-forking model. The model assumes a collection of threads that can be created dynamically and can run asynchronously in parallel. Each thread acts like a standard random-access machine (RAM), with a constant number of shared registers and sharing a common main memory. The model includes a `fork` instruction that forks an asynchronous child thread. A computation starts with a single thread and finishes

when all threads end. In addition to reads and writes to the shared memory, the model includes a test-and-set (TS) instruction. Costs are measured in terms of the work (total number of instructions executed among all threads) and the span (the longest sequence of dependent instructions).

The binary-forking model is meant to capture the performance of algorithms on modern multicore shared-memory machines. Variants of the model have been widely studied [1, 3, 20, 23, 25, 27, 31–35, 41, 42, 45, 50, 54, 89]. They are also widely used in practice, and supported by programming systems such as Cilk [60], the Java fork-join framework [74], X10 [40], Habanero [39], Intel Threading Building Blocks (TBB) [71], and the Microsoft Task Parallel Library [91].

The binary forking model and variants are practical on multicore shared-memory machines in part because they are mostly asynchronous, and in part due to the dynamic binary forking. Asynchrony is important because the processors (cores) on modern machines are themselves highly asynchronous, due to varying delays from cache misses, processor pipelines, branch prediction, hyper-threading, changing clock speeds, interrupts, the operating system scheduler, and several other factors. Binary forking is important since it allows for efficient scheduling in both theory and practice, especially in the asynchronous setting [2, 11, 26, 35]. Efficient scheduling can be achieved even when the number of available processors changes over time [11], which often happens in practice due to shared resources, background jobs, or failed processors.

Due to these considerations, it would seem that these models are more practical for designing parallel algorithms than the more traditional PRAM model [86], which assumes strict synchronization on each step, and a fixed number of processors. One can also argue that they are a more convenient model for designing parallel algorithms, allowing, for example, the easy design of parallel divide-and-conquer algorithms, and avoiding the need to schedule by hand [19]. The PRAM can be simulated on the binary-forking model by forking P threads in a tree for each step of the PRAM. However, this has a $O(\log n)$ overhead in span. This means that algorithms that are optimal on the PRAM are not necessarily optimal when mapped to the binary-forking model. For example, Cole's ingenious pipelined merge sort on n keys and processors takes optimal $O(\log n)$ parallel time (span) on the PRAM [43], but requires $O(\log^2 n)$ span in the binary-forking model due to the cost of synchronization. On the other hand a $O(n \log n)$ work and $O(\log n \log \log n)$ span algorithms in the binary-forking model is know [45]. Therefore finding more efficient direct algorithms for the binary-forking model is an interesting problem. Known results are outlined in Section 1.1.

The variants of the binary-forking model differ in how they synchronize. The most common variant is binary fork-join model where every fork corresponds to a later join, and the fork and

Problem		Work	Span
List Contraction	Sec 3	$O(n)$	$O(\log n)^*$
Sorting	Sec 4	$O(n \log n)^\dagger$	$O(\log n)^*$
Semisorting	Sec 4	$O(n)^\dagger$	$O(\log n)^*$
Random Permutation	Sec 6	$O(n)^\dagger$	$O(\log n)^*$
Range Minimum Query	Sec 7	$O(n)$	$O(\log n)$
Tree Contraction	Sec 8	$O(n)$	$O(\log n)^*$
Ordered-Set Operations (Union, Intersect, Diff.)	Sec 5	$O(m \log(\frac{n}{m} + 1))$	$O(\log n)$

Table 1: The bounds of our new algorithms in the binary-forking model. For ordered-set operations, n and $m \leq n$ are sizes of two sets. *: with high probability (*whp*). †: in expectation. Bounds without superscripts are worst-case bounds. The ordered-set algorithms can work in binary-forking model only with join supported (either by using TS or just as a default primitive), and the rest make use of TS.

corresponding joins are properly nested [1, 23, 27, 31, 34, 35, 45, 50]. Other models allow more powerful synchronization primitives [11, 32, 41, 42, 54, 89]. In this paper we allow a test-and-set (TS), which is a memory operation that atomically checks if a memory location is zero, returning the result, and sets it to one. This seems to give some power over the pure fork-join model. We make use of the TS in many of our algorithms. We justify including a TS instruction by noting that all modern multicore hardware includes the instruction. Furthermore all existing theoretical and practical implementations of the fork-join model require the test-and-set, or equivalently powerful operation to implement the join.

In this paper we describe several algorithms for fundamental problems that are optimal in both work and span in the binary-forking model. In particular, we show the following results.

Theorem 1.1 (Main Theorem). *Sorting, semisorting, list/tree contraction, random permutation, ordered-set operations, and range minimum queries can be computed in the binary-forking model with optimal work and span ($O(\log n)$). In many cases the algorithms are randomized, as summarized in Table 1.*

To achieve these bounds, we develop interesting algorithmic approaches. For some of them, we are inspired by recent results on identifying dependences in sequential iterative algorithms [22, 28, 87]. This paper discusses a non-trivial approach to convert the dependence DAG into an algorithm in the binary-forking model while maintaining the span of the algorithm to be the same as the longest chain in the DAG. This leads to particularly simple algorithms, even compared to previous PRAM algorithms whose span is suboptimal when translated to the binary-forking model. For some other algorithms, we use the n^ϵ-way divide-and-conquer scheme. By splitting the problem into n^ϵ sub-problems and solving them in parallel in logarithmic time, we are able to achieve $O(\log n)$ span for the original problem. Our results on ordered sets are the best known (optimal work in the comparison model and $O(\log n)$ span) even when translated to other models such as the PRAM.

We note that for many of the problems we describe, it remains open whether the same bounds can be achieved deterministically, and also whether they can be achieved in the binary fork-join model without a TS. One could argue that avoiding a TS is more elegant.

1.1 Related Work

There have been many existing parallel algorithms designed based on variants of the binary-forking model (e.g., [1, 3, 13, 14, 20, 23, 25, 27, 29–33, 41, 42, 45, 52–54, 89]). Many of the results are in the setting of cache-efficient algorithms. This is because binary forking in conjunction with work-stealing or space-bounded schedulers leads to strong bounds on the number of cache misses on multiprocessors with various cache configurations [1, 23, 42, 45].

In the binary fork-join model, Blelloch et al. [27] give work-efficient $O(\log n)$ span algorithms for prefix sums and merging, and a work-efficient randomized sorting algorithm with $O(\log^{3/2} n)$ span *whp*[1]. Cole and Ramachandran [45] improved this and gave a deterministic algorithm with span $O(\log n \log \log n)$. This is currently the best known result for sorting in the binary fork-join model, without a test-and-set, and also for deterministic sorting even with a test-and-set.

Allowing for more powerful synchronization, Blelloch et al. [31, 32] discussed how to implement futures using the TS instruction, which leads to some low-span binary-forking algorithms Tang et al. [41, 54, 89] described some dynamic programming algorithms, in the setting of cache efficiency. They also use a TS for synchronization. With this they can reduce the span of a variety of algorithms over fork-join computations without the atomic synchronizations. Without considering the additional support for cache efficiency, we believe their model is equivalent to the binary-forking model.

2 MODELS AND SIMULATIONS

Here we describe the binary-forking model and its relationship to more traditional models of parallel computing, including the PRAM and circuit models. The binary-forking model falls into the class of multithreaded models [11, 26, 34, 35, 44]. Multithreaded computational models assume a collection of threads (sometimes called processes or tasks) that can be dynamically created, and generally run asynchronously. Cost is determined in terms of the total work and the computational span (also called depth or critical path length). There are several variants on multithreaded models depending on how many threads can be forked, how they synchronize, and assumptions about how the memory can be accessed. To be concrete, we define a specific model in this paper.

The binary-forking model. The ***binary-forking model*** consists of ***threads*** that share a common memory. Each thread acts like a sequential RAM—it works on a program stored in the shared memory, has a constant number of registers (including a program counter), and has standard RAM instructions (including an end instruction to finish the computation). The binary-forking model extends the RAM with a fork instruction, which forks a *child* thread. We also employ a special end instruction named endall to indicate the completion of the whole computation. The fork instruction sets the first register to zero in the parent (*forking*) thread and to one in the child (*forked*) thread, to distinguish them. Otherwise the states of the threads are identical, including the program counter to the next instruction. As is standard with the sequential RAM [90],

---
[1]We use the term $O(f(n))$ with high probability (*whp* in n to indicate the bound $O(kf(n))$ holds with probability at least $1 - 1/n^k$ for any $k \geq 1$. With clear context we drop "in n".

we assume that for input size n, all memory locations and registers can hold $O(\log n)$ bits.

In addition to reads and writes, we include a `test-and-set` (TS) instruction in the binary-forking model for accessing memory. The TS is an atomic instruction that reads a memory location and if the memory location is **zero**, sets it to **one**, returning **zero**. Otherwise it leaves the value unchanged returning **one**. We note that all currently produced processors support the TS instruction in hardware.

In a binary-forking model, a computation starts with a single **initial** thread and finishes when endall is called. The invocation to an endall can be determined by the algorithm, for example, through using TS instructions (e.g., to implement join instructions, see below). A computation in the binary-forking model can therefore be viewed as a tree where each node is an instruction with the next instruction as a child, and where the fork instruction has two children corresponding to the next instruction of the original forking thread and the first instruction of the forked thread. The root of the tree is the first instruction of the initial thread. We define the *work* of a computation as the size of the tree (total number of instructions) and the *span* as the depth of the tree (longest path of instructions). We assume the results of memory operations are consistent with some total order (linearization) of the instructions that preserves the partial order defined by the tree. For example, a read will return the value of the previous write or TS to the same location in the total order. The choice of total order can affect the results of a program since threads can communicate through the shared memory. In general, therefore, computations are nondeterministic.

To simplify issues of parallel memory allocation we assume there is an `allocate` instruction that takes a positive integer n and allocates a contiguous block of n memory locations, returning a pointer to the block, and a `free` instruction that given a pointer to an allocated block, frees it.

We use $\mathcal{BF}(W(n), S(n))$ to denote the class of algorithms that require $O(W(n))$ work and $O(S(n))$ span for inputs of size n in the binary-forking model. We use $\mathcal{BF}^k$ when $S(n) = O(\log^k(n))$ and $W(n)$ is polynomial in n, and $\mathcal{BF}^*$ when the span is polylogarithmic and the work is polynomial.

The binary-forking model can be extended to support arbitrary-way forking instead of binary. In particular, the fork instruction can take an integer specifying the number of threads to fork, and each forked thread then gets a unique integer identifier in a register. The focus of this paper, however, is on binary forking since there are no known optimal scheduling results for arbitrary-way forking (see below). The model can also be augmented with more powerful atomic memory operation. For instance, some algorithms [3, 13, 14, 52, 53] use compare-and-swap (CAS) in addition to the above-mentioned model. We refer to this model as the binary-forking model with CAS. A TS is sufficient for our algorithms.

Joining. It can be useful to join threads after forking them, and many models support such joining [11, 26, 35, 44]. This can be implemented by adding a join instruction to the binary-forking model. When reaching a join instruction in thread t the *forking thread* t must "wait" until its most recently *forked* child thread t_c ends. Specifically, in the partial order of the tree mentioned above, it means the partial order is augmented with a dependence from the

end instruction of t_c to the join instruction of t. This partial order is now a series-parallel DAG instead of a tree, and the total order has to be consistent with it. As before, the work is the total number of instructions, but now the span is the longest path of instructions in the DAG instead of tree. We call this the *binary fork-join model*.

Joining can easily be implemented in the binary-forking model without a built-in join instruction, but by using the TS instruction. To implement a join, before each fork we initialize a "synchronization" location to **zero**. For the forking and the forked threads, whichever finishes later is responsible for processing the rest of the computation after the join. This is determined by reaching consensus through the synchronization location. When the forking thread T reaches a join it saves its registers and then performs a TS on the corresponding synchronization location. If the TS returns **one**, this means that the other thread has already finished and set it to **one** first, and T can continue to the next instruction in the program. Otherwise, it means that the other thread has not finished yet, and thus T ends because the other thread will take over the rest of the computation later. When the forked thread reaches its end, it also performs a TS on the synchronization location. Similarly, if the TS returns **zero** it ends, otherwise it loads the registers saved by the forking thread, and jumps to the stored program counter. This implementation preserves work and span within a constant factor. By using fork and join one can also simulate a regular parallel for-loop of size n using divide-and-conquer, which takes $\Theta(\log n)$ span to fork and synchronize.

The simulation implies that the binary-forking model is as least as powerful as the binary fork-join model (with or without TS). We note that unlike binary fork-join model, by using a general TS instead of just a join, the parallelism supported by binary-forking model is not necessary nested. We point out that to implement a constant-time join seems to require an operation at least as powerful as TS. In particular reads and writes by themselves are not powerful enough to get consensus among even just two processes in a wait-free manner, and TS is the seems to be the least powerful memory operation that can achieve two process consensus [69]. This suggests a primitive as powerful as TS is necessary to efficiently implement a join on an asynchronous machine since the two joining threads need to agree (reach consensus) on who will run the continuation.

PRAM. For background, we give a brief description of the PRAM model [86]. A PRAM consists of p processors, each a sequential random access machine (RAM), connected to a common shared memory of unbounded size. Processors run synchronously in lock-step. Although processors have their own instruction pointer, in typical algorithms they all run the same program. There are several variants of the model depending on how concurrent accesses to shared memory are handled—e.g., CRCW allows concurrent reads and writes, and EREW requires exclusive reads and writes. For concurrent writes, in this paper we assume an arbitrary element is written (the most standard assumption). A more detailed description of the model and its variants can be found in JáJá's book on parallel algorithms [73]. As with the binary-forking model, we assume that for an input of size n, memory locations and registers contain at most $O(\log n)$ bits. We use $\mathcal{PRAM}(W(n), S(n))$ to indicate PRAM algorithms that run in $O(W(n))$ work (processor-time

product) and $S(n)$ time, $\mathcal{PRAM}^k$ when the time is $O(\log^k n)$, and $\mathcal{PRAM}^*$ when it is polylogarithmic (both with polynomial work).

Relationship to the PRAM. There have been many scheduling results showing how to schedule binary and multiway forking on various machine models [11, 26, 35]. For example, the following theorem can bound the runtime for programs in the binary-forking model on a PRAM.

THEOREM 2.1 ([11, 34]). *Any computation in the binary-forking model that does W work and has S span can be simulated on P processors of a loosely synchronous parallel machine or the CRCW PRAM in*

$$O\left(\frac{W}{P} + S\right)$$

time whp *in* W.

This is asymptotically optimal (modulo randomization) since the simulation must require the maximum of W/P (assuming perfect balance of work) and S (assuming perfect progress along the critical path). The result is based on a work-stealing scheduler. A slight variant of the theorem applies in a more general setting where individual processors can stop and start [11] and P is the average number of processors available.

Importantly, in the other direction, simulating a p-processor PRAM, even the weakest EREW PRAMg requires a $\Theta(\log p)$ factor loss in span on the binary-forking model. This is a lower bound for any simulation that is faithful to the synchronous steps since just forking p parallel instructions (one step on a PRAM) requires at least $\log p$ steps on the binary-forking model.

Relationship to Circuit Models. Beyond the PRAM we can ask about the relationship to circuit models and to bounded space. Here we use $\mathcal{NC}$ for Nick's class, $\mathcal{AC}$ when allowing unbounded in-degree, and $\mathcal{L}$ for logspace [36, 75]. We first note that $\mathcal{NC} = \mathcal{BF}^*$. This follows directly from the PRAM simulations since $\mathcal{NC} = \mathcal{PRAM}^*$ [75]. We also have the following more fine-grained results. We show the proof in the full version of this paper [24].

THEOREM 2.2.
$$\mathcal{NC}^1 \subseteq \mathcal{L} \subseteq \mathcal{BF}^1 \subseteq \mathcal{AC}^1 = \mathcal{PRAM}^1_{CRCW} \subseteq \mathcal{NC}^2$$

3 LIST CONTRACTION

List ranking [9, 12, 46, 72, 73, 75, 83, 84, 93–95] is one of the canonical problems in the study of parallel algorithms. The problem is: given a set of linked lists, compute for each element its position in the list to which it belongs. The problem can be solved by *list contraction*, which contracts a list by following the pointers in the list. After contraction one can rank the list by a second phase that expands it back out. The problem has received considerable attention because of: (1) its fundamental nature as a pointer-based algorithm that seems on the surface to be sequential; and (2) it has many applications as a subroutine in other algorithms. Wyllie [95] first gave an $O(n \log n)$ work and $O(\log n)$ time algorithm for the problem on the PRAM over 40 years ago. This was later improved to a linear work algorithm [47]. Although this problem has been extensively studied, to the best of our knowledge, all existing linear-work algorithms have $\Omega(\log^2 n)$ span in the binary-forking model because they are all round-based algorithms and run in $\Omega(\log n)$ rounds. The main
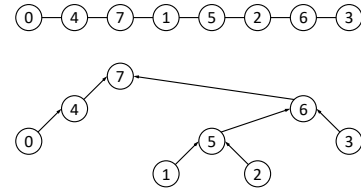
Figure 1: An example of an input list with 8 elements. The number in each element is the priority drawn from a random permutation. The dependences of the contractions are shown as a binary tree structure. In a round-based algorithm [87], the execution is in 4 rounds: $\{0, 1, 2, 3\}$, $\{4, 5\}$, then $\{6\}$, and finally $\{7\}$. In Algorithm 1, the execution is asynchronous, and a possible tree-path decomposition is $\{0, 4\}$, $\{\varnothing\}$, $\{\varnothing\}$, $\{1\}$, $\{\varnothing\}$, $\{2, 5, 6, 7\}$, $\{\varnothing\}$, and $\{3\}$ for all 8 elements from left to right. The length of a tree-path is bounded by the tree height.

result of this section is a randomized, linear work, logarithmic span algorithm in the binary-forking model. Then we also describe how to adapt Wyllie's algorithm to the binary-forking model to achieve $O(n \log n)$ work and $O(\log n)$ span; while not work optimal, this latter algorithm is deterministic. Both algorithms are the first in the binary-forking model to achieve $O(\log n)$ span.

We now present a simple randomized algorithm (Algorithm 1) for list contraction that is theoretically optimal (linear work, and $O(\log n)$ span *whp*) in the binary-forking model. This algorithm is inspired by the list contraction algorithm in [87], but it improves the span by $\Theta(\log n)$, and is quite simple.

The main challenge in designing a work-efficient parallel list contraction algorithm is to avoid simultaneously trying to splice-out two consecutive elements. One solution is via assigning each element a priority from a random permutation. An element can be spliced out only when it has a smaller priority than its previous and next elements, so the neighbor elements cannot be spliced out simultaneously. If the splicing is executed in rounds (namely, splicing out all possible elements in a round-based manner), Shun et al. [87] show that the entire algorithm requires $\Theta(\log n)$ rounds *whp*, leading to $\Theta(\log^2 n)$ span *whp* in the binary-forking model. The dependence structure of the computation is equivalent to a randomized binary tree. On each round we can remove all leaf nodes so the full tree is processed in a number of rounds proportional to the tree depth. An example is illustrated in Figure 1.

After a more careful investigation, we note that the splicing can proceed asynchronously, and not necessarily based on rounds. For example, the last spliced node with priority 7 separates the list into two disjoint sublists, and the contractions on the two sides are independent and can run asynchronously. Conceptually we can do this recursively, and the recursion depth is $\Theta(\log n)$ *whp* [87]. Unfortunately, we cannot directly apply the divide-and-conquer approach since L is stored as a linked list and deciding the elements within sublists is as hard as the list contraction algorithm itself.

We present our algorithm in Algorithm 1. Starting from the leaves, Algorithm 1 performs equivalent steps to the algorithm in [87], but its span is $\Theta(\log n)$ in the binary-forking model; this improvement is achieved by allowing the splicing in each round to run asynchronously. The key idea is that, instead of checking all element for readiness in each round, as long as two children

---

Algorithm 1: LIST-CONTRACTION(L)

Input: A doubly-linked list L of size n. Each element l_i has a random priority ($l_i.p$), next pointer ($l_i.next$), previous pointer ($l_i.prev$) and flag ($l_i.flag$).

1 **parallel foreach** element l_i in L **do**
    // set flag if zero or one child
2    $l_i.flag \leftarrow (pri(l_i) < pri(l_i.prev))$ or $(pri(l_i) < pri(l_i.next))$
3 **parallel foreach** element l_i in L **do**
4    $c \leftarrow l_i$
    // Execute only if c is a leaf node
5   **if** $((pri(c) < pri(c.prev))$ **and** $(pri(c) < pri(c.next)))$ **then**
       // Stop when list is contracted into one node
6   **while not** $(c.prev = null$ **and** $c.next = null)$ **do**
7   Splice c out
8   Let c' be $c.prev$ or $c.next$ with a smaller priority
           // If c is not the last child of c', quit
9   **if** TEST-AND-SET($c'.flag$) **then break**
10         $c \leftarrow c'$

11 **Function** $pri(v)$
12   **if** v = null **then return** ∞ **else return** $v.p$

---

of a node c finished contracting, we trigger c to start contracting immediately. The child of c that finished later is responsible for take over c, and thus can start immediately. In particular, in the algorithm, a parallel-for loop (Line 3) generates n tasks (threads) each for a node in the list. The loop can be implemented by binary forking for $\log_2 n$ levels. Only leaf nodes start the execution, and non-leaf nodes quit immediately (Line 5. These leaves will splice themselves out (Line 7), and then try to move upward and splice its parent (Line 8). We note that a node c cannot be contracted until both of its children have been spliced out. Thus we make the child of c that finishes its splicing later to take over c. This is achieved by letting the two children compete through TEST-AND-SET the *flag* field in c (Line 9). Whichever arrives later takes over and contracts the parent c' (Line 10), and the first one simply terminates its computation (Line 9) and let the second one to take continuation. As an example in Figure 1, the threads for nodes 1 and 2 will both try to work on node 5 after they finish their first splicings. They will both attempt to TEST-AND-SET the *flag* of node 5. The one coming first succeeds and terminates, and the later one will fail and continue splicing node 5. We initialize the *flag* for each node to be 0, except for those with 0 or 1 child (Line 2), for which we set *flag* directly to 1 (they do not need to wait for two children).

THEOREM 3.1. *Algorithm 1 for list contraction does $O(n)$ work (worst case) and has $O(\log n)$ span* whp *in n in the binary-forking model.*

PROOF. The correctness of this algorithm can be shown as it applies the same operations as the list contraction algorithm in [87], although Algorithm 1 runs in a much less synchronous manner. The execution of each thread corresponds to a tree-path in the dependence structure starting from a leaf node and ending on either the root or when winning a TEST-AND-SET. A possible decomposition of the example is shown in the caption of Figure 1. This observation also indicates that the number of iterations of the while-loop on

line 9 for any task is $O(\log n)$ *whp*, bounded by the tree height. The span is therefore $O(\log n)$ *whp*. The work is linear because every time Line 7–9 is executed, one element will be spliced out. □

It is worth noting that, even disregarding the improved span for the binary-forking model, we believe this algorithm is conceptually simpler and easier to implement compared to existing linear-work, logarithmic-time PRAM algorithms [9, 48]. Our algorithm requires starting with a random permutation (discussed further in Section 6). We note that it is straightforward to extend the analysis to using integer priorities instead of random permutations, where the integer priorities are chosen independently and uniformly from the range $[1, n^k]$, for $k \geq 2$, with ties broken arbitrarily.

Binary Forking Wyllie. Here we outline a binary forking version of Wyllie's algorithm with $O(n \log n)$ work and $O(\log n)$ span, both in the worst case. It is useful for our simulation of logspace in $\mathcal{BF}^1$. The idea is to allocate an array of $\log_2 n$ cells per node of the list, each containing two pointers and a boolean value used for TS. At the end of the algorithm the two pointers in the i-th cell (level) will point to the element 2^i links forward and 2^i backward in the list (or a null pointer if fewer than 2^i before or after). The algorithm initially forks off a thread for each node at level 1 in the list. A thread is responsible for splicing out its link at the current level. It does this by writing a pointer to the other neighbor to the corresponding pointer cells of its two neighbors (i.e., splicing itself out), then doing a TS on the boolean flag of each neighbor. For each flag on which it gets a 1 (i.e., it is the second thread to write the pointer at this level), it forks a thread to splice out that neighbor at the next level. Since this fork at the next level only occurs on the second update to the node, both links at the next level must already be available. In general, each splicing step may create 0, 1, or 2 child threads, depending on the timing of arrivals at the neighbors. The first and last element in each list must start with its flag set and writes a null pointer to its one neighbor. As in Wyllie's original algorithm, it is easy to keep counts to generate the ranks of each node in a list. The total work is proportional to the number of cells, $O(n \log n)$ since each cell gets processed once. Since each fork corresponds to performing a splice at a strictly higher level, the span is proportional to the number of levels, i.e., $O(\log n)$.

4 SORTING

In this section we discuss optimal parallel algorithms to comparison sort and semisort [92] n elements using $O(n \log n)$ and $O(n)$ expected work respectively, and $O(\log n)$ span *whp*. For comparison sort, the best previous work-efficient result in the binary-forking model requires $O(\log n \log \log n)$ span [45]. In this paper, we discuss a relatively simple algorithm (Algorithm 2) that sorts n elements in $O(n \log n)$ expected work and $O(\log n)$ span *whp*.

Our algorithm, given in Algorithm 2, is based on sample sorting [59]. It runs recursively. In the base case when the subproblem size falls below a constant threshold, it sorts sequentially. Otherwise, for a subproblem of size n, the algorithm selects $n^{1/3} \log_2 n$ samples uniformly at random, and uses the quadratic-work sorting algorithms to sort these samples (i.e., by making all pairwise comparisons). These two steps can be done in $o(n)$ work and $O(\log n)$ span in the binary-forking model. Then the algorithm subselects

---

Algorithm 2: COMPARISON-SORT(A)

1   Let $n = |A|$
2   **if** n is a constant **then**   Sort the base case and **return**
3   Randomly select $n^{1/3} \log_2 n$ samples
4   Use quadratic sorting algorithm to sort the samples
5   Subsample $n^{1/3}$ pivots from the samples
6   Distribute all elements in A to $n^{1/3} + 1$ buckets based on the samples
     (to form a partition of A to $A_0, A_1, \ldots, A_{n^{1/3}}$). If failed, restart from
     Line 3
7   **parallel foreach** $i \leftarrow 0$ to $n^{1/3}$ **do**   COMPARISON-SORT(A_i)

---

every $\log_2 n$-th sample to be a pivot, and use these $n^{1/3}$ pivots to partition all elements into $n^{1/3} + 1$ buckets.

LEMMA 4.1. *In the distribution step on Line 6 in Algorithm 2, the number of elements falling into one bucket is no more than $c_1 r n^{2/3}$ with probability at least $1 - n^{-c_1}$ for certain constant r and any constant $c_1 > 1$.*

This follows from Chernoff bound. The algorithm then allocates $n^{1/3} + 1$ arrays, one per bucket, each with size $2c_1 r n^{2/3}$. Then in parallel, each element uses binary search to decide its corresponding bucket. It then tries to add itself to a random position in the bucket by using a TS on a flag to reserve it. If the TS fails, it tries again since the slot is already taken. We limit the number of retries for each element to be no more than $c_2 \log_2 n$. If any element cannot find an available slot in this number of retries, the algorithm restarts the process from the random-sampling step (Line 3). Otherwise, after all elements are inserted, the algorithm packs the buckets into contiguous elements for input to the next recursive calls.

THEOREM 4.2. *Algorithm 2 sorts n elements in $O(n \log n)$ expected work and $O(\log n)$ span* whp *in the binary-forking model.*

To bound span, we need to consider the number of retries and the cost of each retry along any path to a leaf in the recursion tree. The number of retries is upper bounded by a geometric distribution since each retry is independent, but the probability of that distribution depends on the level of recursion since problem sizes get smaller. Furthermore the span of a try also depends on the level of recursion (it is bounded by $O(\log n_i)$, where n_i is the input size of level i). To help analyze the span, we will use the following Lemma.

LEMMA 4.3. *Let $X_1 \cdots, X_m$ be independent geometric random variables, and X_i has success probability $p_i = 1 - 2^{-k^i}$ where $k > 1$ is a constant. Then $\sum_{i=1}^{m} k^i \cdot X_i \leq O(ck^m)$ holds with probability at least $1 - 2^{-ck^m}$ for any given constant $c \geq 1$.*

PROOF. We view the contribution from each term $k^i \cdot X_i$ to the sum based on a series of independent unbiased coin flips. The term $k^i \cdot X_i$ can be considered as the event that we toss k^i coins simultaneously, and if all k^i coins are heads we charge k^i to the sum and this process repeats (corresponding to the geometric random variable with probability $p_i = 1 - 2^{-k^i}$). However, in this analysis, we toss one coin at a time until we get a tail, and we charge 1 to the sum for each head before the tail. In this way we can only overestimate the sum. Hence, $\sum_{i=1}^{m} k^i \cdot X_i$ can be upper bounded by the number of heads when tossing an unbiased coin until we see

m tails. We use Chernoff bound[2] $\Pr(X \leq (1-\delta)\mu) \leq e^{-\delta^2 \mu/2}$, where X is the sum of indicator random variables, and $\mu = \mathbb{E}[X]$. Now let's consider the probability that we see more than qk^m heads before m tails. Since $m < k^m$, we analyze the probability to see no more than k^m tails, which only increases the probability. In this case, we make $(q + 1)k^m$ tosses, so $\mu = (q + 1)k^m/2$ and $\delta = (q - 1)/(q + 1)$. The probability is therefore no more than:

$$\exp\left[-\left(\frac{q-1}{q+1}\right)^2 \cdot \frac{(q+1)k^m}{4}\right] = \exp\left[-\frac{(q-1)^2 k^m}{4(q+1)}\right]$$

$$= \exp\left[-\frac{(q^2 - 2q + 1)k^m}{4(q+1)}\right] < \exp\left[-\frac{(q^2 - 3q - 4)k^m}{4(q+1)}\right]$$

$$= \exp\left[-\frac{(q+1)(q-4)k^m}{4(q+1)}\right] = e^{-(\frac{q}{4}-1)k^m} < 2^{-(\frac{q}{4}-1)k^m}$$

This proves the lemma by setting $q = 4(c + 1)$.   □

PROOF OF THEOREM 4.2. The main challenge is to analyze the work and span for the distribution cost (Line 6), especially to bound the cost of restarting the distribution step. There are two reasons that the call return to Line 3: badly chosen pivots such that some buckets contain too many elements and become overfull (defined later), or unlucky random number sequences such that the positions tried by a particular element are all occupied (for more than $c_2 \log n$ consecutive slots). We say a bucket is overfull if it has more than $c_1 r n^{2/3}$ elements (more than half of the allocated space). From Lemma 4.1, the probability of this event is no more than n^{-c_1}. We pessimistically assume that the distribution step restarts once a bucket is overfull. Therefore, for the latter case with a bad random number sequence, the allocated array is always no more than half-full, which is useful in analyzing this case.

We now analyze the additional costs for the restarts. For the latter case, with probability at most n^{1-c_2}, at least one element retries more than $c_2 \log_2 n$ times. For the first case, the probability that any bucket is oversize is n^{1-c_1}. By setting c_1 and c_2 to be at least 2, the expected work including restarts is asymptotically bounded by the first round of selecting pivots and distributing the elements. The work of the first round is bounded by $O(n \log n)$ since there are n elements, each doing a binary search and then each trying at most $O(\log n)$ locations. Therefore the expected work for each distribution is $O(n_i \log n_i)$, where n_i is the size of the input.

The total number of elements across any level of recursion is at most n since every element goes to at most one bucket. Also the size of each input reduces to at most $kn^{2/3}$ from level to level, for some constant k. The total expected work across each level of recursion therefore decreases geometrically from level to level. Hence the total work is asymptotically bounded by the work at the root of the recursion, which is $O(n \log n)$ in expectation.

We now focus on the span, and first analyze the case for the chain of subproblems for one element. The number of recursive levels is $O(\log \log n)$. For each level with subproblem size n', let $c = c_1 = c_2 \geq 2$. The probability for a restart is less than $2(n')^{1-c}$, and the span cost for a restart is $c \log_2 n$. Treating the number of restarts in each level as a random variable, we can plug in Lemma 4.3 with $k = 1.5$ and $m = \log_{1.5} \log_2 n$, and show that the span of this chain is $O(ck^m) = O(c \log n)$ with probability at least $1 - 2^{-c \log n} =$

---

[2]https://en.wikipedia.org/wiki/Chernoff_bound.

$1 - n^{-c}$. Then by taking a union bound for the n chains to all leaves of the recursion, the probability is at least $1 - n^{1-c}$. Combining the analyses of the work and the span proves the theorem. □

Semisorting. Semisorting reorders an input array of n keys such that equal keys are contiguous but different keys are not necessarily in sorted order. It can be used to implement integer sort with a small key range. Semisorting is a widely-used primitive in parallel algorithms (e.g., the random permutation algorithm in Section 6).

We note that with the new comparison sorting algorithm with optimal work and span, we can plug it in the semisorting algorithm by Gu et al. [65] (Step 3 in Algorithm 1). The rest of the algorithm is similar to the distribution step but just run for one round, so it naturally fits in the binary-forking model with no additional cost. Hence, this randomized algorithm is optimal in the binary-forking model—$O(n)$ expected work and $O(\log n)$ span *whp*.

5 ORDERED SET-SET OPERATIONS

In this section, we show deterministic algorithms for ordered set-set operations (UNION, INTERSECTION and DIFFERENCE) based on weight-balanced binary search trees. In particular we prove the following theorem.

THEOREM 5.1. UNION, INTERSECTION *and* DIFFERENCE *of two ordered sets of size n and $m < n$ can be solved in* $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$ *work and* $O(\log n)$ *span in the binary-forking model. This is optimal for comparison-based algorithms.*

Our approach is based on a (roughly $\sqrt{n}$-way) divide-and-conquer algorithm with lazy reconstruction-based rebalancing. At a high-level, for two sets of size n and m ($\le n$), we will split both trees with $d - 1$ pivots equally distributed among the $m + n$ elements, where $d = \Theta(\sqrt{m + n})$ is a power of 2. The algorithm runs recursively until the base case when $m' \le \sqrt{m' + n'}$, where m' and n' are the sizes of the two input trees in the current recursive call, respectively. For the base cases, we apply a weaker (work-inefficient) algorithm discussed in the full version of this paper [24]. The work-inefficient approach will not affect the overall asymptotic bound because of the criterion at which the base cases are reached. After that, the d pieces are connected using the pivots. At this time, rebalancing may occur, but we do not handle it immediately. Instead, we apply a final step at the end of the algorithm to recursively rebalance the output tree based on a reconstruction-based algorithm discussed in Section 5.4. The high-level idea is that, whenever a subtree has two children imbalanced by more than some constant factor (i.e., one subtree is much larger than the other one), the whole subtree gets flattened and reconstructed. Otherwise, the subtree can be rebalanced using a constant number of rotations. An illustration of our algorithm is shown in Figure 2. Due to page limitation, we put the algorithm description of base case algorithms, and the cost analysis of the algorithm in the full version of the paper [24], and only briefly show some intuition in Section 5.5.

5.1 Background and Related Work

Ordered set-set operations UNION, INTERSECTION and DIFFERENCE are fundamental algorithmic primitives, and there is a rich literature of efficient algorithms to implement them. For two ordered sets of size n and $m \le n$, the lower bound on the number of comparisons

(and hence work or sequential time) is $\Omega\left(m \log\left(\frac{n}{m} + 1\right)\right)$ [70]. The lower bound on span in the binary-forking model is $\Omega(\log n)$. Many sequential and parallel algorithms match the work bound [5, 21, 31, 38]. In the parallel setting, some algorithms achieve $O(\log n)$ span on the PRAM [80, 81]. However, they are not work-efficient, requiring $O(m \log n)$ work. There is also previous work focusing on I/O efficiency [15] and concurrent operations [37, 57] for parallel trees, and parallel data structures supporting batches [4, 64, 78].

Some previous algorithms achieve optimal work and polylogarithmic span. Blelloch and Reid-Miller proposed algorithms on treaps with optimal expected work and $O(\log n)$ span *whp* on an EREW PRAM with scan operations, which translates to $O(\log^2 n)$ span in the binary-forking model. Akhremtsev and Sanders [5] described an algorithm for array-tree UNION based on (a, b)-trees with optimal work and $O(\log n)$ span on a CRCW PRAM. Blelloch et al. [31] proposed ordered set algorithms for a variety of balancing schemes [21] with optimal work. All the above-mentioned algorithms have $O(\log m \log n)$ span in the binary-forking model. There have also been parallel bulk operations for self-adjusting data structures [4]. As far as we know, there is no parallel algorithm for ordered set functions (UNION, INTERSECTION and DIFFERENCE) with optimal work and $O(\log n)$ span in the binary-forking model.

5.2 Preliminaries

Given a totally ordered universe U, the problem is to take the union, intersection, and difference of two subsets of U. We assume the comparison model over the *elements* of U, and require that the inputs and outputs can be enumerated in-order with no additional comparison (i.e., no cheating by being lazy).

We assume the two inputs have sizes m and $n \ge m$ stored in weight-balanced binary trees [79] with balancing parameter α (WBB[α] tree). The weight of a subtree is defined as its size plus one, such that the weight of a tree node is always the sum of the weights of its two children. WBB[α] trees maintain the invariant that for any two subtrees of a node, the weights are within a factor of α ($0 < \alpha \le 1 - 1/\sqrt{2}$) of each other. For the two input trees, we refer to the tree of size n as the *large tree*, denoted as T_L, and the tree of size m as the *smaller tree*, denoted as T_S. We present two definitions as follows. Note that these two definitions are more general than the definitions of ancestors and descendants, *since k may or may not appear in T*.

DEFINITION 1. *In a tree T, the* upper nodes *of an element $k \in U$, are all the nodes in T on the search path to k (inclusive).*

DEFINITION 2. *In a tree T, an element $k \in U$ falls into a subtree $T_x \in T$, if the search path to k in T overlaps the subtree T_x.*

Persistent Data Structures. In this section, we use underlying *persistent* [55] (and actually purely functional) tree structure, which uses path-copying to update the weight-balanced trees. This means that when a change is made to a node v in the tree, a copy of the path to v is made, leaving the old path and old value of v intact. Figure 3 shows an example of inserting a new element into the tree. Such a persistent insertion algorithm also copies nodes that are involved in rotations since their child pointers change.

In particular, our algorithm will use a persistent SPLIT(T, k) function on WBB[α] trees as discussed in [21, 88]. This function splits
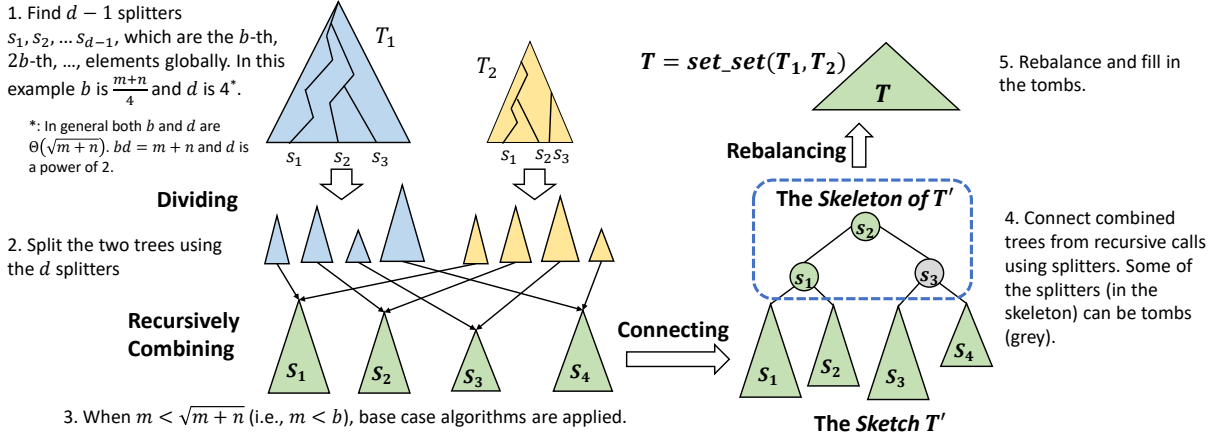
Figure 2: An illustration of the set-set algorithms. We first split both trees into chunks by the glaobally b-th, $2b$-th, ..., elements. Here $b = \frac{m+n}{4}$, but in general $b = (n + m)/d$ should be $\Theta(\sqrt{m+n})$ where $d = \Theta(\sqrt{m+n})$ is a power of 2. We then recursively sketch each pair of chunks, until we reach the base case and call the base case algorithms. We then connect the results with pivots, and get the *sketch* of the result tree. Finally we rebalance the tree structure and fill in all tombs.
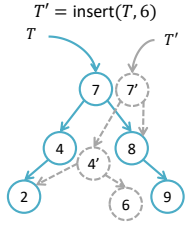
Figure 3: A persistent insertion on a tree. The algorithm basically copies all tree nodes on the insertion path, such that the new (copied) root represents the output tree, and the input tree is intact represented by the old root pointer. This algorithm costs $O(\log n)$ time for an input tree of size n.

Algorithm 3: $T \leftarrow$ SET_SET(T_1, T_2), the main algorithm for ordered set-set operations

---

Input: Two weight-balanced trees storing two ordered sets.
Output: A weight-balanced tree T storing the
   union/intersection/difference of the two input sets.

1 **if** $|T_1| < |T_2|$ **then return** SET_SET(T_2, T_1)
2 $T' \leftarrow$ SKETCH(T_1, T_2) // Algorithm 4
3 $T \leftarrow$ REBALANCE(T', FALSE) // Algorithm 5
4 **return** T

---

tree T by key k into two trees and a bit, such that all keys smaller than k and larger than k will be stored the two output trees, respectively, and the bit indicates if $k \in T$. Because of path-copying, the persistent SPLIT returns two output trees and leaves the input tree intact. This algorithm costs $O(\log n)$ work on a tree of size n.

5.3 The Main Algorithms

We first give a high-level description of our algorithms for the three set-set functions. As mentioned, we denote the larger input tree as T_L, and the smaller input tree as T_S. We use two steps, ***sketching*** and ***rebalancing***. The sketching step aims at combining the elements in the two input trees in-order into one tree, which is not necessarily balanced. The rebalancing step will apply a top-down algorithm to rebalance the whole tree by the WBB[α] criteria.

Our sketching algorithm is based on a d-way divide-and-conquer scheme, where $d = \Theta(\sqrt{n+m})$ is a power of 2. It is a recursive

Algorithm 4: $T' \leftarrow$ SKETCH(T_1, T_2)

Input: Two WBB[α] trees. T_1 is from the original larger tree T_L and
   T_2 is from the original smaller tree T_S.
Output: A binary tree sketch T' representing the
   union/intersection/difference of the two input sets.

1 Let $n' \leftarrow |T_1|$ and $m' \leftarrow |T_2|$
2 **if** n' is 0 **then return** T_2
3 **if** m' is 0 **then return** T_1
4 **if** $m' < \sqrt{n' + m'}$ **then return** BASE_CASE(T_1, T_2)
5 $d \leftarrow 2^{\lceil \log_2 \sqrt{m'+n'} \rceil}$
6 $b \leftarrow (m' + n')/d$
7 Let $splitter_0 \leftarrow -\infty$ and $splitter_d \leftarrow +\infty$
8 **parallel for** $i \leftarrow 1$ to $d - 1$ **do**
9   Find $splitter_i$, which is the $(i \cdot b)$-th element in T_1 and T_2
        (duplicate value counts twice) by dual-binary search
10   Let f_i indicate if $splitter_i$ is a tomb
11 **parallel for** $i \leftarrow 1$ to d **do**
12   Split T_1 using $splitter_{i-1}$ and $splitter_i$, output tree $T_{1,i}$
13   Split T_2 using $splitter_{i-1}$ and $splitter_i$, output tree $T_{2,i}$
14    $T'_i \leftarrow$ SKETCH$(T_{1,i}, T_{2,i})$
15 Connect $T'_1, \ldots, T'_d$ using $splitter_1, \ldots, splitter_{d-1}$
16 **return** the result tree

---

algorithm, for which the two input trees are denoted as T_1 and T_2. In particular, T_1 contains a subset of T_L and T_2 contains a subset of T_S. The algorithm will combine the two subsets and return one result tree. Note that even though $T_1 \subseteq T_L$ and $T_2 \subseteq T_S$, the sizes of T_1 is not necessarily larger than T_2. Throughout the recursive process, we track the following quantities for each tree node v:

   (1) The size of the subtree, noted as size(v).
   (2) The number of elements originally from T_L, noted as large(v).
   (3) The number of elements originally from T_S, noted as small(v).
   (4) The number of elements appearing both in T_L and T_S, noted as common(v).

Algorithm 5: $\langle T, e \rangle \leftarrow$ Rebalance(T', last)

Input: A tree sketch T', and a boolean flag *last* indicating if the last element should be extracted.

Output: A valid weight-balanced tree T with no tombs. If *last* is true, e is the last element extracted from T'.

Note: EffectiveSize(T) returns the number of non-tombs in T.

1   **if** EffectiveSize(T') is 0 **then return** $\langle \varnothing, \varnothing \rangle$
2   **if** T' is obtained by base cases (Line 4 in Algorithm 4) **then**
3   **if** last **then return** $\langle$RemoveLast(T'), Last(T')$\rangle$
4   **else return** $\langle T', \varnothing \rangle$
5   **if** no nodes in T_S fall into T' (by checking small(T')) **then**
6   **if** last **then return** $\langle$RemoveLast(T'), Last(T')$\rangle$
7   **else return** $\langle T', \varnothing \rangle$
8   **if** last **then** $b \leftarrow 1$ **else** $b \leftarrow 0$
9   **if** EffectiveSize(LeftTree(T')) + 1 and EffectiveSize(RightTree(T')) − b + 1 differs by more than a factor of $2/\alpha$ **then**
10   Flatten T' and reconstruct it (if last then extract the last element in T')
11   **return** the new tree
12   **if** the root of T' is a tomb **then** $t \leftarrow$ true **else** $t \leftarrow$ false
13   **In parallel:**
14      $\langle T_l, e_l \rangle$ = Rebalance(LeftTree(T'), t)
15      $\langle T_r, e_r \rangle$ = Rebalance(RightTree(T'), last)
16   **if** EffectiveSize(RightTree(T')) is 0 **and** last **then**
17      $T \leftarrow T_l$
18   **if** the root of T' is a tomb **then** $e \leftarrow e_l$
19   **else** $e \leftarrow T'.root$
20   **else**
21      $e \leftarrow e_r$
22   **if** the root of T' is a tomb **then** $T \leftarrow$ Connect(T_l, e_l, T_r)
23   **else** $T \leftarrow$ Connect(T_l, $T'.root$, T_r)
24   **return** $\langle$RebalanceByRoatation(T), $e \rangle$

The tree size size(v) is required by the WBB[α] invariant. The other three are used for $T_L - T_S$, $T_S - T_L$, and $T_S \cap T_L$ respectively. The generic algorithm for all three operations is given in Algorithms 3, 4, and 5. An illustration is shown in Figure 2. The difference between the three set-set functions is only in the base cases.

We now present the two steps of the algorithm in details:

(1) **Sketching** (Algorithm 4). This step generates an output tree T' with all elements in the result, although not rebalanced. There are three subcomponents in this step. Denote $n' = |T_1|$ and $m' = |T_2|$, which means the number of tree nodes handled by this recursive call that are originally from the larger and smaller tree, respectively. As mentioned, m' can be even larger than n' in some of the recursive calls.

  (a) **Base Case.** When $m' < \sqrt{n' + m'}$, the algorithm reaches the base case. It calls the work-inefficient algorithm to generate a balanced output tree in $O(m' \log n')$ work and $O(\log n')$ span, which are presented in the full version of this paper [24].

  (b) **Dividing.** We then use $d - 1$ **pivots** to split both input trees into d **chunks**, and denote the partitioning of T_1 as $T_{1,\{1,\ldots,d\}}$, and T_2 as $T_{2,\{1,\ldots,d\}}$. The $d - 1$ pivots are the global b-th, $2b$-th, $\ldots$ elements in the two trees, where $b = (n + m)/d$, so that $|T_{1,i}| + |T_{2,i}|$ for all i have the same value (or differ by

at most 1). All the splits (Line 12-13) can be done in parallel using a persistent split algorithm on weight-balanced trees [21]. We then apply the algorithm recursively on each pair of chunks.

Note that not all the pivots should appear in the output tree of the entire algorithm, depending on the set function. For example, for Intersection, those pivots that only appear in one tree will not show up at the end. In this case, in the Sketch step, we will mark such pivot nodes as *tombs*, and filter them out later in the rebalancing step.

  (c) **Connecting.** After the dividing substep and recursive calls, we have $d - 1$ pivots (including tombs), and d combined chunks returned by the recursive calls. In the connecting substep, we directly connect them regardless of balance. Since d is a power of 2, the $d - 1$ pivots will form a full balanced binary tree structure on the top $\log_2 d$ levels, and all the chunks output from recursive calls will dangle on the d pivots. This process is shown in Line 15.

The output T' of the Sketch step is a binary tree, which may or may not be balanced. We will call T' the **sketch** of the final output of the algorithm. We also call the top $\log_2 d$ levels in T' consists of pivots the **skeleton** of T'. We note that T' may contain tombs, and we will filter them out in the next step.

(2) **Rebalancing**. We will use a reconstruction-based rebalancing algorithm to remove the tombs and rebalance the sketch tree T' (Algorithm 5, see more details in Section 5.4). This rebalancing algorithm is stand-alone, and is of independent interest.

5.4   The Rebalancing Algorithm

We now present the reconstruction-based rebalancing algorithm. A similar idea was also used in [29]. In this paper, we use this technique to support better parallelism instead of write-efficiency.

We use the *effective size* of a subtree as the number of elements in this subtree excluding all tombs. The effective size for a tree node v can be computed based on size(v), large(v), small(v) and common(v), depending on the specific set operation. It is used to determine if two subtrees will be balanced after removing all tombs.

The rebalancing algorithm is given in Algorithm 5. The algorithm recursively *settles* each level top-down. For a tree node, we check the effective sizes of its two children and decide if they are almost-balanced. Here *almost-balanced* indicates that sizes of the two subtrees differ by at most a factor of $2/\alpha$.[3] If not, we flatten the subtree and re-build it. Otherwise, we recursively settle its two children, and after that we re-connect the two subtrees back and rebalance using at most a constant number of rotations.

We also need to filter out *tombs*, since they should not appear in the output tree. We do this recursively. If the current subtree root of T' in Algorithm 5 is a tomb, we will need to fill it in using the last element in its left subtree. We note that the effective size of the left subtree cannot be 0 (otherwise the algorithm returns at Line 11). To do this, the algorithm will take an extra boolean argument last denoting if the last element of the result needs to be extracted (returned as e in the output of Algorithm 5). In this case, if the root

---

[3]Generally speaking, the constant 2 here can be any value, but here we use 2 for convenience.

of T' is a tomb, the algorithm simply passes a `true` value to the left recursive call, getting the last element to replace the tomb.

For computing the last value (denoted as r), there are two cases. First, if the subtree needs rebalancing, then after flattening the elements into an array, we simply take out the last element in the array as r and return. Extracting the last element is inlined in the process of reconstruction (Line 10). Otherwise, we recursively deal with the two subtrees. If `last` is true, we also extract the last element in its right subtree.

Multiple base cases apply to this rebalancing algorithm. If the effective size of T' is 0, the algorithm directly returns an empty tree and an empty element. The second case is when no element in T_S falls into T'. This can be determined by looking at $small(T')$. Note that all the chunks in the sketching algorithm is designed to be the same size. Therefore, in this case, the whole subtree should be (almost) perfectly balanced, so we directly return it. These base cases are essential in bounding the work of rebalancing, since we do not need to traverse the whole subtree for these special cases.

5.5 Base Case Algorithms and Cost Analysis

Due to page limitation, we put the algorithm description of base case algorithms and the cost analysis of the algorithm in the full version of this paper [24]. Here we show a very brief description about the intuition.

Base case algorithms. The base case algorithms $\textsc{Base\_Case}(T_1, T_2)$ in Algorithm 4 use an work-inefficient version ($O(m' \log n')$ work and $O(\log(n' + m'))$ span) of the set-set algorithms. These algorithms are applied when $m' < \sqrt{n' + m'}$, which guarantees the total base case cost is $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$. The intuition of the base case algorithms is to search all m' elements from T_1 in T_2, and based on the set operation being performed, add (remove) the m' elements into (from) T_2. The same rebalancing algorithm as in Section 5.4 is applied to guarantee a balanced output tree. Detailed description is in the full version of this paper [24].

Span. We will show that all base cases, $\textsc{Sketch}$, and $\textsc{Rebalance}$ algorithms have span $O(\log(n + m))$. We first prove that the height of the sketch T' is $O(\log(n + m))$ in the full version of this paper [24]. The span of the base cases is straight-forward. For $\textsc{Sketch}$, this bound holds because of the $\sqrt{m + n}$-way divide-and-conquer. For $\textsc{Rebalance}$, the span holds because the algorithm settles each node top-down, and settling each level in the skeleton only costs a constant span. For the skeleton of the returned tree, if a node is nearly-balanced, then a constant number of rotations settles it. Otherwise, flattening and reconstructing a tree of height h takes $O(h)$ span, which is also equivalent to a constant per level. In all, the span is $O(\log(n + m))$. We formally prove the span of the algorithm in the full paper [24].

Work. For work, we will prove that all base cases, $\textsc{Sketch}$, and $\textsc{Rebalance}$ algorithms cost work $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$. We first show that the number of pivots is $O(m)$. Most interestingly, for $\textsc{Rebalance}$, the optimality in work lies in the reconstruction-based algorithm. For all pivots in the skeleton, if it is nearly balanced, the rebalancing cost is a constant. Therefore the total work is proportional to the size of the skeleton, which is no more than $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$.

To show the total reconstruction work, in the sketch T', we mark all upper nodes of the elements in T_S as red. There are at most $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$ red nodes in T'. We will show that the reconstruction work averaged to each red node is a constant. The key observation is that, rebalancing for a subtree $T_x \in T'$ happens only when there are $m_x \geq c|T_x|$ red nodes in T_x, where c is a constant. This is because the two subtrees of T_x are supposed to have the same size ($\sqrt{n' + m'}$) due to the selection of pivots. However there can be duplicates in $\textsc{Union}$; also $\textsc{Intersection}$ and $\textsc{Difference}$ do not keep all input elements in the output. Therefore there can be imbalanced in size. We will show that the size of either subtree changes by no more than m_x. Therefore, to make them unbalanced, m_x has to be at least $c|T_x|$ for some constant c. This makes the average cost per red node to be $O(1)$. Adding the cost of all red nodes gives the stated optimal work bound. We will formally prove the work of the algorithm in the full version of this paper [24].

6 RANDOM PERMUTATION

Generating random permutation in parallel is useful in parallel algorithms, and is used in the list and tree contraction algorithms in this paper. Hence it has been well-studied both theoretically [6, 8, 51, 61–63, 66, 68, 77, 82, 87] and experimentally [49, 67, 87]. To the best of our knowledge, none of these algorithms can be implemented in the binary-forking model using linear work and $O(\log n)$ span. We now consider the simple sequential algorithm of Knuth [76] (Durstenfeld's [56]) shuffle that iteratively decides each element:

---

1 **Function** $\textsc{KnuthShuffle}(A, H)$
2 $\quad$ $A[i] \leftarrow i$ for all $i = 0, \ldots, n - 1$
3 $\quad$ **for** $i \leftarrow n - 1$ to 0 **do**
4 $\quad\quad$ swap($A[i], A[H[i]]$)

---

where $H[i]$ is an integer uniformly drawn between 0 and $i - 1$, and $A[\cdot]$ is the output random permutation.

A recent paper [87] shows that this sequential iterative algorithm is readily parallel. The key idea is to apply multiple swaps in parallel as long as the sets of source and destination locations of the swaps are disjoint. Figure 4 shows an example, and we can swap location 5 and 2, 7 and 1, 6 and 3 simultaneously in the first round, and the three swaps do not interfere each other. If the nodes pointing to the same node are chained together and the self-loops are removed, we get the dependences of the computation. An example is given in Figure 4(b). Similar to list contraction, we can execute the swaps for all leaf nodes and remove them from the tree in a round-based manner. It can be shown that the modified dependences by chaining all the roots in the dependence forest (as shown in Figure 4(c)) correspond to a random binary search tree, and the tree depth is again bounded by $O(\log n)$ *whp*. The span of this algorithm is therefore $O(\log^2 n)$ *whp* in the binary-forking model.

Similar to the new list contraction algorithm discussed in Section 3, the computation can be executed asynchronously. Namely, the swaps in different leaves or subtrees are independent. Therefore, once the dependence structure is generated, we can apply a similar approach as in Algorithm 1, but instead of splicing out each node, we swap the values for the pair of nodes.
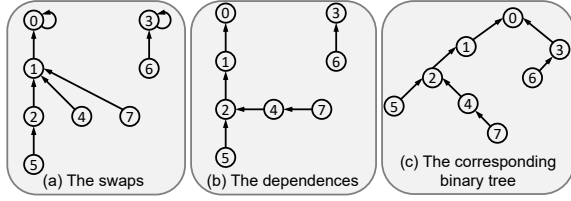
Figure 4: An example when $H = [0, 0, 1, 3, 1, 2, 3, 1]$**. Figure (a) indicates the destinations of the swaps shown by** H**. The dependences of the swaps are shown by Figure (b), indicating the order of the swaps. Figure (c) links the roots of the forest to make it a binary tree.**

The remaining question is how to generate the dependence structure. We do this in two steps. We first semisort all nodes based on the destination locations (grouping the nodes on all the horizontal chains in Figure 4(b) or right chains in Figure 4(c)). Then we use an algorithm that takes quadratic work to sort the nodes within each group, and connect the nodes as discussed.

THEOREM 6.1. *The above algorithm generates a random permutation of size n using $O(n)$ expected work and $O(\log n)$ span* whp *in the binary-forking model.*

PROOF. Similar to the list contraction algorithm in Section 3, this algorithm applies the same operations as the random permutation algorithm in [87], and the swaps obey the same ordering for any pair of nodes with dependency. The improvement for span is due to allowing asynchrony for disjoint subtrees.

The cost after the construction of dependence tree is the same as the list contraction algorithm (Algorithm 1), which is $O(n)$ work and $O(\log n)$ span *whp*. For constructing the dependence tree, the semisort step takes $O(n)$ expected work and $O(\log n)$ span *whp* using the algorithm in Section 4. The quadratic work sorting can easily be implemented in $O(\log n)$ span *whp*, as in Section 4. We now analyze the work to sort the chains.

Let a 0/1 random variable $A_{i,j}$ is 1 if $H[i] = j$ for $j < i$, and the probability $\Pr[A_{i,j} = 1]$ is $1/i$. $\Pr[A_{i,j}A_{k,j} = 1]$ is then $1/(ik)$ for $j < i < k$ since they are independent. The expected overall work for sorting is (omitting constant in front of $A_{i,j}$).

$$\mathbb{E}[W_{RandPerm}(n)] = \mathbb{E}\left[\sum_{j=1}^{n}\left(\sum_{i=j}^{n}A_{i,j}\right)^2\right]$$

$$= \mathbb{E}\left[\sum_{j=1}^{n}\sum_{i=j}^{n}A_{i,j}^2\right] + 2 \cdot \mathbb{E}\left[\sum_{j=1}^{n}\sum_{i=j+1}^{n}\sum_{k=i+1}^{n}A_{i,j}A_{k,j}\right]$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{i-1}1/i + \sum_{k=1}^{n}\sum_{i=1}^{k-1}\sum_{j=1}^{i-1}1/ik = O(n)$$

Combining all results gives the stated theorem. ☐

7 RANGE MINIMUM QUERIES

Given an array A of size n, the range minimum query (RMQ) takes two input indices i and j, and reports the minimal value within this range. RMQ is a fundamental algorithmic building block that is used to solve other problems such as the lowest common ancestor

(LCA) problem on rooted trees, the longest common prefix (LCP) problem, and lots of other problems on trees, strings and graphs.

An optimal RMQ algorithm requires linear preprocessing work and space, and constant query time. It can be achieved by a variety of algorithms (e.g., [7, 10, 16, 17, 58]). These algorithms are based on the data structure referred to as the sparse table that can be precomputed in $O(n \log n)$ work where n is the input size, and constant RMQ cost. To further improve the work, the high-level idea in these algorithms is to chunk the array into $O(n/\log n)$ groups each with size $O(\log n)$, find the minima of the groups, and only preprocess the sparse table for the minima. Within each group, these algorithms use different techniques to preprocess in $O(\log n)$ work per group, and support constant query cost within each group. Then for a range minimum query (i, j), the minimum can be answered by combining by the query for the sparse table for the whole groups in this range, and the query for the boundary groups that contain elements indexed at i and j. These algorithms can be trivially parallelized in the PRAM model using $O(\log n)$ span (time), but when translating to the binary-forking model, the span becomes $O(\log^2 n)$ in preprocessing the sparse table, and needs to be improved.

For simplicity, we first assume the number of groups n' is a power of 2. In the classic sparse table, we denote $T_{i,k}$ as the minimal value between group range i and $i + 2^k - 1$. It can be computed as $\min\left\{T_{i,k-1}, T_{i+2^{k-1},k-1}\right\}$. Let $k = \lfloor\log_2(j-i)\rfloor$. Then for query from group i to $j(> i)$, we have $RMQ(i, j) = \min\left\{T_{i,k}, T_{j-2^k+1,k}\right\}$. Directly parallelizing the construction for the sparse table uses $O(\log^2 n)$ span—$O(\log n)$ levels in total and $O(\log n)$ span within each level. We now consider a variant of the sparse table which is easier to be generated in the binary-forking model and equivalently effective.
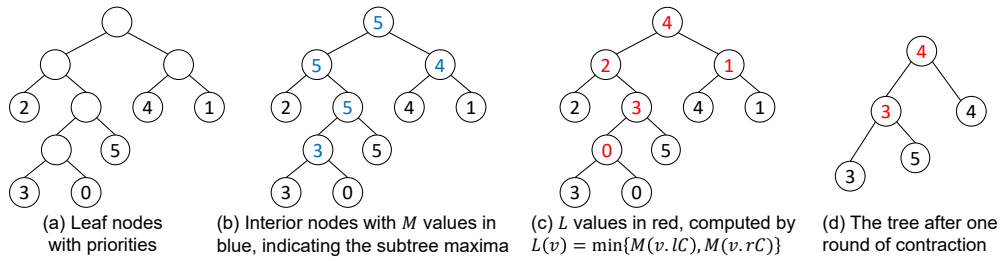
In the modified version, we similarly have $\log_2 n'$ levels, and in k-th level we partition the array into $n'/2^k$ subarrays each with size 2^k. For each subarray, we further partition it to two parts with equal size, and compute the suffix minima for the left side, and prefix minima for the right side. We denote $T'_{i,k}$ as such value with index i in the k-th level. For each query (i, j), we find the highest significant bit that is different for i and $j \geq i$. If this bit is the k-th bit from the right, then we have $RMQ(i, j) = \min\left\{T'_{i,k}, T'_{j,k}\right\}$. An illustration is shown in Figure 6.

We now describe how to compute $T'_{i,k}$. We note that the computation for each subarray is independent, and each takes linear work and logarithmic span proportional to the subarray size [18]. Since each element corresponds to $\log_2 n'$ computed values, the overall cost is therefore $O(n' \cdot \log_2 n') = O(n)$ work and $O(\log n)$ span.

THEOREM 7.1. *The range minimum queries for an array of size n can be preprocessed in $O(n)$ work and $O(\log n)$ span in the binary-forking model, and each query requires constant cost.*

8 TREE CONTRACTION

Parallel algorithms for tree contraction have received considerable interest because of its ample applications for many tree and graph applications [13, 73, 77, 85, 87]. There are many variants of parallel tree contraction. Here we will assume we are contracting rooted binary trees in which every internal node has exactly two children.

(a) Leaf nodes with priorities

(b) Interior nodes with M values in blue, indicating the subtree maxima

(c) L values in red, computed by $L(v) = \min\{M(v.lC), M(v.rC)\}$

(d) The tree after one round of contraction

Figure 5: An example of the tree contraction algorithm. It first generates the priority of the leaf node from a random permutation, as shown in Figure (a). Then every interior node computes the highest priority label in the subtree, and the computed values are shown as the blue numbers in Figure (b). Then each interior node v is paired with the leaf node that has the priority which is stored in the child node different from the priority of v. Taking the root node as an example, the priority 5 is different from the right node which has priority 4, so the root node is paired with the leaf node with priority 4. The pairing results are shown as the red numbers in Figure (c). In this example, leaf nodes 0, 1, and 2 can be contracted simultaneously, and the contracted tree is shown in Figure (d).
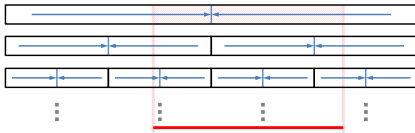
Figure 6: An illustration of the modified sparse table in Section 7. The range is hierarchically partitioned into logarithmic number of levels, and the prefix and suffix minima are computed as the arrows indicate. For each query range shown as the red segment, we can locate a unique level such that the minimum of the range can be answered by the suffix and the prefix minima (the shaded range).

Any rooted tree can be reformatted to this shape in linear work and logarithmic span. We assume the tree T has n leaf nodes (and $n-1$ interior nodes). We use $v.lC$ and $v.rC$ to denote the left and the right child of a node v, respectively.

List contraction can be considered as a degenerated case of tree contraction when all interior nodes are chained up. As a result, we do not know an optimal parallel algorithm for tree contraction with $O(n)$ work and $O(\log n)$ span. Similarly, the difficulty in designing such an algorithm remains in using no synchronization.

Here we consider parallelizing the sequential iterative algorithm that "rakes" one leaf node at a time. A rake operation removes a leaf node and its parent node v, and if v is not the root, it sets the other child of v to replace v as the child of v's parent. We assign each leaf node a priority drawn from a random permutation, so the priority defines a global ordering of the nodes to be removed, and eventually only one node with the lowest priority remains. By maintaining some additional information on the tree nodes, we can apply a variety of tree operations such as expression evaluation, roofix or leafix, which are useful in many applications [85].

Similar to list contraction, we want to avoid applying two rake operations simultaneously such that one of the interior nodes is the parent of the other. Beyond that, we can rake a set of leaf nodes in parallel. For instance, in Figure 5(a), we can contract leaf nodes 0, 1, 2, and their parents together, as shown in Figure 5(d).

To decide the nodes that can be processed together, we define $M(v)$ of each interior node v as the lowest priority (maximum value) of any of the leaves in its subtree (blue numbers in Figure 5(b)). Based on $M(\cdot)$, we further define $L(v) = \min\{M(v.lC), v.rC)\}$ (red

numbers in Figure 5(c)) if v is an interior node, or its own priority if v is a leaf node. $L(v)$ defines a one-to-one mapping between the interior nodes and the leaf nodes (except for one leaf node that stays at the end), and $L(v) = u$ indicates that v will be raked by the leaf node u. Based on the labeling, the parallel algorithm in [87] checks every node v, and it can be raked immediately if v' parent has an L value smaller than those of v's sibling and v's grandparent (if it exists). Otherwise the node waits for the next round. If we rake all possible leaf nodes in a round-based manner, the number of rounds is $O(\log n)$ *whp*, leading to an $O(\log^2 n)$ span *whp* in the binary-forking model.

Assuming $L(\cdot)$ has already been computed, we can change the round-based algorithm to an asynchronous divide-and-conquer algorithm similar to the list contraction algorithm (Algorithm 1) in Section 3. The only difference is when setting the *flag*s, since now there can be 1, 2, or 3 directions that may activate a postponed node (in list contraction it is either 1 or 2, depending on the initialization of the *flag* array). This however, can be easily decided by checking the number of neighbor interior nodes. Similarly, the last thread corresponding to the contraction of a neighbor node that reaches a postponed node activates it and apply the rake operation. Since the longest possible path has length $O(\log n)$, the algorithm for the contraction phase uses $O(n)$ work, and $O(\log n)$ span *whp*.

The last challenge is computing $L(\cdot)$. As shown in Figure 5(b), computing $M(\cdot)$ is a leafix operation on the tree (analogy to prefix minima but from the leaves to the root), which can be solved by the standard range minimum queries as discussed in Section 7, based on Euler-tour of the input tree. In Section 3, we discussed the list ranking algorithm to generate the Euler tour. As a result, computing $M(\cdot)$ and $L(\cdot)$ uses $O(n)$ work and $O(\log n)$ span *whp*. In summary, we have the following theorem.

THEOREM 8.1. *Tree contraction uses $O(n)$ work and $O(\log n)$ span* whp *in the binary-forking model.*

REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theoretical Computer Science (TCS)* 35, 3 (2002).

[2] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 769–782.

[3] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably Good Scheduling for Parallel Programs That Use Data Structures Through Implicit Batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[4] Kunal Agrawal, Seth Gilbert, and Wei Quan Lim. 2018. Parallel Working-Set Search Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[5] Yaroslav Akhremtsev and Peter Sanders. 2016. Fast Parallel Operations on Search Trees. In *IEEE International Conference on High Performance Computing (HiPC)*.

[6] Laurent Alonso and Ren Schott. 1996. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science (TCS)* 159, 1 (1996).

[7] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. 2004. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems (TOCS)* 37, 3 (2004), 441–456.

[8] Richard J. Anderson. 1990. Parallel Algorithms for Generating Random Permutations on a Shared Memory Machine. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[9] Richard J Anderson and Gary L Miller. 1990. A simple randomized parallel algorithm for list-ranking. *Inform. Process. Lett.* 33, 5 (1990), 269–273.

[10] Lars Arge, Johannes Fischer, Peter Sanders, and Nodari Sitchinava. 2013. On (dynamic) range minimum queries in external memory. In *Workshop on Algorithms and Data Structures*. Springer, 37–48.

[11] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[12] Sara Baase. 1993. Introduction to Parallel Connectivity, List Ranking, and Euler Tour Techniques. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann, 61–114.

[13] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2016. Parallel Algorithms for Asymmetric Read-Write Costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[14] Naama Ben-David, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. Implicit Decomposition for Write-Efficient Connectivity Algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[15] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, et al. 2019. Small refinements to the DAM can have big consequences for data-structure design. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 265–274.

[16] Michael A. Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics (LATIN)*. Springer, 88–94.

[17] Omer Berkman and Uzi Vishkin. 1993. Recursive star-tree parallel data structure. *SIAM J. Scientific Computing* 22, 2 (1993), 221–242.

[18] Guy E. Blelloch. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann.

[19] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (March 1996).

[20] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[21] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[22] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[23] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[24] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2019. Optimal parallel algorithms in the binary-forking model. In *arXiv preprint:1903.04650*.

[25] Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[26] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably Efficient Scheduling for Languages with Fine-grained Parallelism. *J. ACM* 46, 2 (March 1999).

[27] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[28] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2016. Parallelism in Randomized Incremental Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[29] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[30] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Randomized Incremental Convex Hull is Highly Parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[31] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[32] Guy E. Blelloch and Margaret Reid-Miller. 1999. Pipelining with futures. *Theory of Computing Systems (TOCS)* 32, 3 (1999).

[33] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[34] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Scientific Computing* 27, 1 (1998).

[35] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[36] Allan Borodin. 1977. On Relating Time and Space to Size and Depth. *SIAM J. Scientific Computing* 6, 4 (1977), 733–744.

[37] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+ tree. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 58–67.

[38] Mark R. Brown and Robert Endre Tarjan. 1980. Design and Analysis of a Data Structure for Representing Sorted Lists. *SIAM J. Scientific Computing* 9, 3 (1980), 594–614.

[39] Zoran Budimlić, Vincent Cavé, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Jisheng Zhao, and Vivek Sarkar. 2011. The design and implementation of the habanero-java parallel programming language. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*. 185–186.

[40] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 519–538.

[41] Rezaul Chowdhury, Pramod Ganapathi, Yuan Tang, and Jesmin Jahan Tithi. 2017. Provably Efficient Scheduling of Cache-oblivious Wavefront Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 339–350.

[42] Rezaul A. Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. 2013. Oblivious algorithms for multicores and networks of processors. *J. Parallel and Distrib. Comput.* 73, 7 (2013), 911–925.

[43] Richard Cole. 1988. Parallel Merge Sort. *SIAM J. Scientific Computing* 17, 4 (1988).

[44] Richard Cole and Vijaya Ramachandran. 2017. Bounding Cache Miss Costs of Multithreaded Computations Under General Schedulers: Extended Abstract. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[45] Richard Cole and Vijaya Ramachandran. 2017. Resource Oblivious Sorting on Multicores. *ACM Transactions on Parallel Computing (TOPC)* 3, 4 (2017).

[46] Richard Cole and Uzi Vishkin. 1986. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *ACM Symposium on Theory of Computing (STOC)*. 206–219.

[47] Richard Cole and Uzi Vishkin. 1988. Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM J. Scientific Computing* 17 (1988), 128–142.

[48] Richard Cole and Uzi Vishkin. 1989. Faster optimal parallel prefix sums and list ranking. *Information and computation* 81, 3 (1989), 334–352.

[49] Guojing Cong and David A. Bader. 2005. An Empirical Analysis of Parallel Random Permutation Algorithms ON SMPs. In *Parallel and Distributed Computing and Systems (PDCS)*.

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

[51] Artur Czumaj, Przemyslawa Kanarek, Miroslaw Kutylowski, and Krzysztof Lorys. 1998. Fast generation of random permutations via networks simulation. *Algorithmica* (1998).

[52] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[53] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. 2020. Semi-Asymmetric Parallel Graph Algorithms for NVRAMs. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020).

[54] David Dinh, Harsha Vardhan Simhadri, and Yuan Tang. 2016. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[55] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making data structures persistent. *J. Computer and System Sciences* 38, 1 (1989),

86–124.

[56] Richard Durstenfeld. 1964. Algorithm 235: Random Permutation. *Commun. ACM* 7, 7 (1964), 420.

[57] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 275–286.

[58] Johannes Fischer and Volker Heun. 2006. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Symposium on Combinatorial Pattern Matching (CPM)*. Springer, 36–48.

[59] W Donald Frazer and Archie C McKellar. 1970. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM* 17, 3 (1970), 496–507.

[60] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 33, 5 (1998), 212–223.

[61] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. 1996. Efficient Low-Contention Parallel Algorithms. *J. Computer and System Sciences* 53, 3 (1996).

[62] Joseph Gil. 1991. Fast load balancing on a PRAM. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[63] Joseph Gil, Yossi Matias, and Uzi Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.

[64] Seth Gilbert and Wei Quan Lim. 2019. Parallel Finger Search Structures. *arXiv preprint arXiv:1908.02741* (2019).

[65] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. 2015. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[66] Jens Gustedt. 2003. Randomized permutations in a coarse grained parallel environment. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[67] Jens Gustedt. 2008. Engineering Parallel In-place Random Generation of Integer Permutations. In *Workshop on Experimental Algorithmics*.

[68] Torben Hagerup. 1991. Fast parallel generation of random permutations. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. Springer.

[69] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124âĂŞ149.

[70] Frank K. Hwang and Shen Lin. 1972. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. Scientific Computing* 1, 1 (1972), 31–39.

[71] Intel Threading Building Blocks [n.d.]. https://www.threadingbuildingblocks.org.

[72] Riko Jacob, Tobias Lieber, and Nodari Sitchinava. 2014. On the complexity of list ranking in the parallel external memory model. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 384–395.

[73] J. JaJa. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[74] Java Fork-Join [n.d.]. http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[75] Richard M. Karp and Vijaya Ramachandran. 1990. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press.

[76] Donald E. Knuth. 1969. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley.

[77] Gary L. Miller and John H. Reif. 1985. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 478–489.

[78] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. Bq: A lock-free queue with batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 99–109.

[79] Jürg Nievergelt and Edward M Reingold. 1973. Binary search trees of bounded balance. *SIAM J. Scientific Computing* 2, 1 (1973).

[80] Heejin Park and Kunsoo Park. 2001. Parallel Algorithms for Red-Black Trees. *Theoretical Computer Science (TCS)* 262, 1–2 (2001), 415–435.

[81] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. 1983. Parallel Dictionaries in 2-3 Trees. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 597–609.

[82] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Scientific Computing* 18, 3 (1989).

[83] Abhiram Ranade. 1998. A simple optimal list ranking algorithm. In *IEEE International Conference on High Performance Computing (HiPC)*.

[84] Margaret Reid-Miller, Gary L. Miller, and Francesmary Modugno. 1993. List Ranking and Parallel Tree Contraction. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann, 115–194.

[85] John H. Reif. 1993. *Synthesis of Paralell Algorithms*. Morgan Kaufmann.

[86] Yossi Shiloach and Uzi Vishkin. 1981. Finding the Maximum, Merging, and Sorting in a Parallel Computation Model. *J. Algorithms* 2, 1 (1981).

[87] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T Fineman, and Phillip B Gibbons. 2015. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.

[88] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[89] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A Chowdhury. 2015. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[90] Robert Endre Tarjan. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[91] Task Parallel Library (TPL) [n.d.]. https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx.

[92] Leslie G. Valiant. 1990. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science (Vol. A)*, Jan van Leeuwen (Ed.). MIT Press, 943–973.

[93] Uzi Vishkin. 1984. Randomized Speed-Ups in Parallel Computation. In *ACM Symposium on Theory of Computing (STOC)*. 230–239.

[94] Uzi Vishkin. 1993. Advanced Parallel Prefix-sums, List Ranking and Connectivity. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann, 215–257.

[95] James C. Wyllie. 1979. *The Complexity of Parallel Computations*. Technical Report TR-79-387. Department of Computer Science, Cornell University, Ithaca, NY.