Data Oblivious Algorithms for Multicores

Vijaya Ramachandran
vlr@cs.utexas.edu
University of Texas at Austin
Austin, TX, U.S.A.

Elaine Shi
runting@gmail.com
Carnegie Mellon University
Pittsburgh, PA, U.S.A.

ABSTRACT

A data-oblivious algorithm is an algorithm whose memory access pattern is independent of the input values. We initiate the study of parallel data oblivious algorithms on realistic multicores, best captured by the binary fork-join model of computation. We present a data-oblivious CREW binary fork-join sorting algorithm with optimal total work and optimal (cache-oblivious) cache complexity, and in $O(\log n \log\log n)$ span (i.e., parallel time); these bounds match the best-known bounds for binary fork-join cache-efficient insecure algorithms. Using our sorting algorithm as a core primitive, we show how to data-obliviously simulate general PRAM algorithms in the binary fork-join model with non-trivial efficiency, and we present data-oblivious algorithms for several applications including list ranking, Euler tour, tree contraction, connected components, and minimum spanning forest. All of our data oblivious algorithms have bounds that either match or improve over the best known bounds for insecure algorithms.

Complementing these asymptotically efficient results, we present a practical variant of our sorting algorithm that is self-contained and potentially implementable. It has optimal caching cost, and it is only a $\log\log n$ factor off from optimal work and about a $\log n$ factor off in terms of span. We also present an EREW variant with optimal work and caching cost, and with the same asymptotic span.

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**; • **Security and privacy → Cryptography**.

KEYWORDS

data-oblivious algorithms; multithreaded algorithms; CREW binary fork-join

ACM Reference Format:
Vijaya Ramachandran and Elaine Shi. 2021. Data Oblivious Algorithms for Multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3409964.3461783

This work was supported in part by grants NSF CCF-2008241, 1601879 and 2128519, ONR YIP, and a Packard Fellowship.

SPAA '21, July 6–8, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461783

1 INTRODUCTION

As secure processors such as Intel SGX (with hyperthreading) become widely adopted, there is a growing appetite for private analytics on big data. Most prior works on data-oblivious algorithms adopt the classical PRAM model to capture parallelism. However, it is widely understood that PRAM does not best capture realistic multicore processors, nor does it reflect parallel programming models adopted in practice.

We initiate the study of *data-oblivious* algorithms for a multicore architecture where parallelism and synchronization are expressed with nested binary fork-join operations. Imagine that a client outsources encrypted data to an untrusted cloud server which is equipped with a secure, multicore processor architecture (e.g., Intel SGX with hyperthreading). All data contents are encrypted to the secure processor's secret key both at rest and in transit. Data is decrypted only inside the secure cores' hardware sandboxes where computation takes place. However, it is well-known that encryption alone does not guarantee privacy, since access patterns to even encrypted data leak a lot of sensitive information [41, 54]. To defend against access pattern leakage, an active line of work [15, 17, 36, 37, 45, 52] has focused on how to design algorithms whose access pattern distributions do not depend on the secret inputs — such algorithms are called *data-oblivious* algorithms, and are the focus of our work. In this paper we present nested fork-join data-oblivious algorithms for several fundamental problems that are highly parallel, and work- and cache-efficient. Throughout this paper, we consider only data oblivious algorithms that are unconditionally secure (often called "statistically secure"), i.e., without the need to make any computational hardness assumptions.

There has been some prior work exploring the design of *parallel* data-oblivious algorithms. Most of these prior parallel data-oblivious algorithms [15, 17, 47] adopted PRAM as the model of computation. However, the global synchronization between cores at each parallel step of the computation in a PRAM algorithm does not best capture modern multicore architectures, where the cores typically proceed asynchronously. To better account for synchronization cost, a long line of work [1, 3, 8–14, 22–24, 26–30, 34] has adopted a multithreaded computation model with CREW (Concurrent Read Exclusive Write) shared memory in which parallelism is expressed through paired fork and join operations. A binary fork spawns two tasks that can execute in parallel. Its corresponding join is a synchronization point: both of the spawned tasks must complete before the computation can proceed beyond this join. Such a binary fork-join model is also adopted in practice, and supported by programming systems such as Cilk [33], the Java fork-join framework [48], X10 [20], Habanero [16], Intel Threading Building Blocks [35], and the Microsoft Task Parallel Library [46].

Efficiency in a multi-threaded algorithm with binary fork-joins is measured through the following metrics: the algorithm's *total work*

(i.e., the sequential execution time), its *cache complexity* (i.e., the number of cache misses), and its *span* (i.e., the length of the longest path in the computation DAG). The span is also the number of parallel steps in the computation assuming that unlimited number of processors are available and they all execute at the same rate. We discuss the binary fork-join model in more detail in Section 1.2.

1.1 Our Results

We present highly parallel data-oblivious multithreaded algorithms that are also cache-efficient for a suite of fundamental computation tasks, starting with sorting. Our results show how to get privacy for free for a range of tasks in this important parallel computation model. Further, all of our algorithms are cache-agnostic [32][1], i.e., the algorithm need not know the cache's parameters including the cache-line (i.e., block) size and cache size. Besides devising new algorithms, our work also makes a conceptual contribution by creating a bridge between two lines of work from the cryptography and algorithms literature, respectively. We now state our results.

Sorting. To attain our results, the most important building block is data-oblivious sorting. We present a randomized data-oblivious, cache-agnostic CREW binary fork-join sorting algorithm, Butterfly-Sort, which matches the work, span and cache-oblivious caching bounds of SPMS [30], the current best insecure algorithm. These bounds are given in Theorem 1.1. As noted in [30], these bounds are optimal for work and cache complexity, and are within an $O(\log\log n)$ factor of optimality for span (even for the CREW PRAM). The key ingredient in Butterfly-Sort is Butterfly-Random-Permute (B-RPermute), which randomly permutes the input array without leaking the permutation, and guided by a butterfly network. The overall Butterfly-Sort simply runs B-RPermute and then runs SPMS on the randomly permuted array.

In comparison with known cache-agnostic and data-oblivious sorting algorithms [19], Butterfly-Sort improves the span by an (almost) exponential factor — the prior work [19] requires n^{ϵ} parallel runtime for some constant $\epsilon \in (0,1)$ even without the binary fork-join constraint. We now state our sorting result.

Theorem 1.1 (Butterfly-Sort). *Let B denote the block size and M denote the cache size. Under the standard tall cache assumption $M = \Omega(B^{1+\epsilon})$, and $M = \Omega(\log^{1+\epsilon} n)$ where $\epsilon \in (0,1)$ is an arbitrarily small constant,* Butterfly-Sort *is a cache-agnostic CREW binary fork-join algorithm that obliviously sorts an array of size n with an optimal cache complexity of $O((n/B)\cdot\log_M n)$, optimal total work $O(n\log n)$, and $O(\log n\cdot\log\log n)$ span which matches the best known non-oblivious algorithm in the same model.*

Practical and EREW variants. We devise a conceptually simple algorithm, Butterfly-Random-Sort (B-RSort), to sort an input array *that has been randomly permuted.* This algorithm uses a collection of pivots, and is similar in structure to our B-RPermute algorithm. By replacing SPMS with B-RSort to sort the randomly permuted output of B-RPermute, we obtain a simple oblivious scheme for sorting, which we call Butterfly-Butterfly-Sort (BB-Sort). By varying the primitives used within BB-Sort we obtain two useful versions: a potentially practical sorting algorithm which uses bitonic sort for small sub-problems, and an efficient EREW binary fork-join version which retains AKS sorting for small subproblems.

Our practical version uses bitonic sort as the main primitive, and for this we present an EREW binary fork-join bitonic sort algorithm that improves on the naïve binary fork-join version by achieving span $O(\log^2 n\cdot\log\log n)$ and cache-agnostic caching cost $O((n/B)\cdot\log_M n\cdot\log(n/M))$ while retaining its $O(n\log^2 n)$ work.

The use of AKS in our EREW and CREW algorithms may appear impractical. However, it is to be observed that prior $O(n\log n)$-work oblivious algorithms for sorting — the AKS network [2], Zigzag sort [38], and an $O(n\log n)$ version of oblivious bucket sort [4] — all use expanders within their construction. Sorting can be performed using an efficient oblivious priority queue that does not use expanders [42, 51]; however this incurs $\omega(n\log n)$ work to achieve a negligible in n failure probability. Further, this method is not cache-efficient and is inherently sequential. It is to be noted that our practical version does not use AKS or expanders.

Table 3 in Section 3 lists the bounds for our sorting algorithms.

Data-oblivious simulation of PRAM in CREW binary fork-join. Using our sorting algorithm, we show how to compile any CRCW PRAM algorithm to a data-oblivious, cache-agnostic binary-fork algorithm with non-trivial efficiency. We present two results along these lines. The first is a compiler that works efficiently for space-bounded PRAM programs, i.e., when the space s used is close to the number of processors p. We argue that this is an important special case because our space-bounded simulation of CRCW PRAM on oblivious binary fork-join (*space-bounded PRAM on OBFJ*) allows us to derive oblivious binary fork-join algorithms for several computational tasks that are cornerstones of the parallel algorithms literature. Our space-bounded PRAM on OBFJ result is given in Theorem 4.1 in Section 4. As stated there, each step of a p-processor CRCW PRAM can be emulated within the work, span, and cache-agnostic caching bounds for sorting $O(p)$ elements.

Our second PRAM on OBFJ result works for the general case when the space s consumed by the PRAM can be much greater than (e.g., a polynomial function in) the number of processors p. For this setting, we show a result that strictly generalizes the best known Oblivious Parallel RAM (OPRAM) construction [17, 18]. Specifically, we can compile any CRCW PRAM to a data oblivious, cache-agnostic, binary fork-join program where each parallel step in the original PRAM can be simulated with $p\log^2 s$ total work and $O(\log s\cdot\log\log s)$ span. The exact statement is in Theorem 4.2 in Section 4.2. In terms of work and span, the bounds in this result match the asymptotical performance of the best prior OPRAM result [17, 18], which is a PRAM on oblivious PRAM result, and hence would only imply results for unbounded forking.[2]. Moreover, we also give explicit cache complexity bounds for our oblivious simulation which was not considered in prior work [17, 18].

To obtain the above PRAM simulation results, we use some building blocks that are core to the oblivious algorithms literature, called *aggregation, propagation,* and *send-receive* [15, 17, 47]. Table 2

[1] "Cache-agnostic" is also called cache-oblivious in the algorithms literature [32]. In order to avoid confusion with data obliviousness, we will reserve the term 'oblivious' for data-oblivious, and we will use *cache-agnostic* in place of cache-oblivious.

[2] The concurrent work by Asharov et al. [6] shows that assuming the existence of one-way functions, each parallel step of a CRCW PRAM can be obliviously simulated in $O(\log s)$ work and $O(\log s)$ parallel time on a *CRCW* PRAM. Their work is of a different nature because they consider computational security and moreover, their target PRAM allows concurrent writes.

Table 1: Comparison with prior insecure algorithms. $\widetilde{O}(\cdot)$ hides a single $\log\log n$ factor. LR = "list ranking", ET-Tree = "Tree computations with Euler tour", TC = "Tree contraction", CC = "connected components", MSF = "minimum spanning forest". For graph problems, n is the number of vertices, and $m = \Omega(n)$ is the number of edges. We compare with insecure cache-efficient CREW binary fork-join algorithms. The prior bounds, except for tree contraction, are from [26], and implicit in other work [12, 24]. The prior result for sort is SPMS sort [25, 30]. The prior bound for tree contraction (TC) is from [12]. A '†' next to a result indicates that we improve the performance relative to the best known bound for the insecure case. If cache-efficiency is not considered for insecure algorithms, the best CREW binary fork-join span for all tasks except Sort is $O(\log^2 n)$; for Sort it remains $\widetilde{O}(\log n)$.

Task	Our data-oblivious algorithm			Previous best insecure algorithm		
	work	span	cache	work	span	cache
Sort	$O(n \log n)$	$\widetilde{O}(\log n)$	$O(\frac{n}{B} \log_M n)$	$O(n \log n)$	$\widetilde{O}(\log n)$	$O(\frac{n}{B} \log_M n)$
LR	$O(n \log n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n)$	$O(n \log n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n)$
ET-Tree	$O(n \log n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n)$	$O(n \log n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n)$
TC†	$O(n \log n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n)$	$O(n \log n)$	$\widetilde{O}(\log^3 n)$	$O(\frac{n}{B} \log_M n)$
CC†	$O(m \log^2 n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{m}{B} \log_M n \log n)$	$O(m \log^2 n)$	$\widetilde{O}(\log^3 n)$	$O(\frac{m}{B} \log_M n \log n)$
MSF†	$O(m \log^2 n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{m}{B} \log_M n \log n)$	$O(m \log^2 n)$	$\widetilde{O}(\log^3 n)$	$O(\frac{m}{B} \log_M n \log n)$

Table 2: Comparison with prior oblivious algorithms. The prior best is obtained by taking the best known oblivious PRAM algorithm and naïvely fork and join n threads at every PRAM step. $\widetilde{O}$ hides a single $\log\log n$ or $\log\log s$ factor. Aggr = aggregation, Prop = propagation, S-R = send-receive, PRAM = oblivious simulation of a p-processor, s-space PRAM (cost of simulating a single step, assuming $s \geq p$), and $\bigstar = O(\log s \cdot ((p/B) \cdot \log_M p + p \cdot \log_B s))$. The best known algorithm for aggregation and propagation are due to [17, 47], send-receive is obtained by combining [17, 47], [2], and [19], oblivious simulation of PRAM is due to or implied by [15, 18].

Obliv. Alg.	Our algorithm			Prior best		
	work	span	cache	work	span	cache
Aggr	$O(n)$	$O(\log n)$	$O(n/B)$	$O(n)$	$O(\log^2 n)$	$O(n/B)$
Prop	$O(n)$	$O(\log n)$	$O(n/B)$	$O(n)$	$O(\log^2 n)$	$O(n/B)$
Sort & S-R	$O(n \log n)$	$\widetilde{O}(\log n)$	$O(\frac{n}{B} \log_M n)$	$O(n \log n)$ $O(n \log n \log^2 \log n)$	$O(\log^2 n)$ $O(n^\epsilon)$	$O(n \log n)$ $O(\frac{n}{B} \log_M n)$
PRAM	$O(s \log s)$ $O(p \log^2 s)$	$\widetilde{O}(\log s)$ $\widetilde{O}(\log s)$	$O(\frac{s}{B} \log_M s)$ $\bigstar$ in caption	$O(s \log^2 s)$ $O(p \log^2 s)$	$O(\log^2 s)$ $\widetilde{O}(\log^2 s)$	$O(s \log s)$ $O(p \log^2 s)$

shows the performance bounds of our algorithms for these building blocks and general PRAM simulation vs. the best known *data-oblivious* results, where the latter is obtained by taking the best known oblivious PRAM algorithm and naïvely forking n threads in a binary-tree fashion for every PRAM step. The table shows that we improve over the prior best results for all tasks.

Applications. There is a very large collection of efficient PRAM algorithms in the literature for a variety of important problems. We use our PRAM simulation results as a generic way of translating some of these algorithms into efficient data-oblivious algorithms in binary fork-join. For other important computational problems we use our new sorting algorithm as a building block to directly obtain efficient data-oblivious algorithms.

We list our results in Table 1 and compare with results for insecure cache-efficient CREW binary fork-join algorithms. Our results for tree contraction (TC), connected components (CC), and minimum spanning forest (MSF) are obtained through our space-bounded PRAM on OBFJ result, and in all three cases our data-oblivious binary fork-join algorithms improve the commonly-cited span by a $\log n$ factor while matching the total work and cache complexity of the best known insecure algorithms. For other computational tasks such as list ranking and rooted tree computations with Euler tour, we design data-oblivious algorithms that match the best work, span and cache-efficiency bounds known for insecure algorithms in the binary fork-join model: these give better bounds than what we would achieve with the PRAM simulation.

Throughout, we allow our algorithms to have only an extremely small failure probability (either in correctness or in security) that is $o(1/n^k)$, for any constant k. We refer to such a probability as being *negligible* in n. Such strong bounds are typical for cryptographic and security applications, and are stronger than the w.h.p. bounds (failure probability $O(1/n^k)$, for any constant k) commonly used for standard (insecure) randomized algorithms.

Road-map. The rest of the paper is organized as follows. The rest of this section has short backgrounds on multithreaded, cache-efficient and data-oblivious algorithms. Section 2 is on Butterfly-Random-Permute and Butterfly-Sort, our most efficient data oblivious sorting algorithm. Section 3 is on Butterfly-Random-Sort which gives our simple oblivious algorithm for sorting a random permutation, and Butterfly-Butterfly-Sort which gives our practical and EREW sorting algorithms. Section 4 has PRAM simulations, and Section 5 has applications. Many details are in the online full version [50].

1.2 Binary Fork-Join Model

For parallelism we consider a multicore environment consisting of P cores (or processors), each with a private cache of size M. The processors communicate through an arbitrarily large shared memory. Data is organized in blocks (or 'cache lines') of size B.

We will express parallelism using a multithreaded model through paired fork and join operations (See Chapter 27 in Cormen et al. [31] for an introduction to this model). The execution of a binary fork-join algorithm starts at a single processor. A thread can rely on a *fork* operation to spawn two tasks; and the forked parallel tasks can be executed by other processors, as determined by a scheduler. The two tasks will later *join*, and the corresponding join serves as a synchronization point: both of the spawned tasks must complete before the computation can proceed beyond this join. The memory accesses are CREW: concurrent reads are allowed at a memory location but no concurrent writes at a location.

The *work* of a multithreaded algorithm Alg is the total number of operations it executes; equivalently, it is the sequential time when it is executed on one processor. The *span* of Alg, often called its critical path length or parallel time T_∞, is the number of parallel steps in the algorithm. The cache complexity of a binary fork-join algorithm is the total number of cache misses incurred across all processors during the execution.

For cache-efficiency, we will use the cache-oblivious model in Frigo et al. [32]. As noted earlier, in this paper we will use the term *cache-agnostic* in place of cache-oblivious, and reserve the term 'oblivious' for data-obliviousness. We assume a cache of size M partitioned into blocks (or cache-lines) of size B. We will use $Q_{sort}(n) = \Theta((n/B)\cdot\log_M n))$ to denote the optimal caching bound for sorting n elements, and for a cache-agnostic algorithm to achieve this bound we need $M = \Omega(B^2)$ (or $M = \Omega(B^{1+\delta})$ for some given arbitrarily small constant $\delta > 0$). Several sorting algorithms with optimal work (i.e., sequential time) and cache-agnostic caching cost are known, including two in [32].

We would like to design binary fork-join algorithms with small work, small sequential caching cost, preferably cache-agnostic, and small span. As explained in our online full version [50], this will lead to good parallelism and cache-efficiency in an execution under randomized work stealing [14].

Let $T_{\text{sort}}(n)$ be the smallest span for a binary fork-join algorithm that sorts n elements with a comparison-based sort. It is readily seen that $T_{\text{sort}}(n) = \Omega(\log n)$. The current best binary fork-join algorithm for sorting is SPMS (Sample Partition Merge Sort) [30]. This algorithm has span $O(\log n \cdot \log\log n)$ with optimal work $W_{\text{sort}} = O(n \log n)$ and optimal cache complexity $Q_{\text{sort}}(n)$. But the SPMS algorithm is not data-oblivious. No non-trivial data oblivious parallel sorting algorithm with optimal cache complexity was known prior to the results we present in this paper.

More background on caching and multithreaded computations is given in our online full version [50].

1.3 Data Oblivious Binary Fork-Join Algorithms

Unlike the terminology "cache obliviousness", data obliviousness captures the privacy requirement of a program. As mentioned in the introduction, we consider a multicore secure processor like Intel's SGX (with hyperthreading), and all data is encrypted to the secure processor's secret key at rest or in transit. Therefore, the adversary, e.g., a compromised operating system or a malicious system administrator with physical access to the machine, cannot observe the contents of memory. However, the adversary may control the scheduler that schedules threads onto cores. Moreover, it can observe 1) the computation DAG that captures the pattern of forks and joins, and 2) the memory addresses accessed by all threads of the binary fork-join program. The above observations jointly form the "access patterns" of the binary fork-join program.

We adapt the standard definition of data obliviousness [17, 36, 37] to the binary fork-join setting.

Definition 1.2 (Data oblivious binary fork-join algorithm). We say that a binary fork-join algorithm Alg data-obliviously realizes (or obliviously realizes) a possibly randomized functionality $\mathcal{F}$, iff there exists a simulator Sim such that for any input $\mathbf{I}$, the following two distributions have negligible statistical distance: 1) the joint distribution of Alg's output on input $\mathbf{I}$ and the access patterns; and 2) $(\mathcal{F}(\mathbf{I}), \mathsf{Sim}(|\mathbf{I}|))$.

Note that the simulator Sim knows only the length of the input $\mathbf{I}$ but not the contents of $\mathbf{I}$, i.e., the access patterns are simulatable without knowing the input.

We stress that our notion of data obliviousness continues to hold if the adversary can observe the exact address a processor accesses, even if that data is in cache. In this way, our security notion can rule out attacks that try to glean secret information through many types of cache-timing attacks. Our notion secures even against computationally unbounded adversaries, often referred to as *statistical security* in the cryptography literature.

One sometimes inefficient way to design binary-fork-join algorithms is to take a Concurrent-Read-Exclusive-Write (CREW) PRAM algorithm, and simply fork n threads in a binary-tree fashion to simulate every step of the PRAM. If the original PRAM has $T(n)$ parallel runtime and $W(n)$ work, then the same program has $T(n)\cdot\log n$ span and $W(n)$ work in a binary-fork-join model. Moreover, if the algorithm obliviously realizes some functionality $\mathcal{F}$ on a CREW PRAM, the same algorithm obliviously realizes $\mathcal{F}$ in a binary-fork-join model too. More details on data obliviousness are in our online full version [50].

2 BUTTERFLY-SORT

To sort the input array, our CREW binary fork-join algorithm Butterfly-Sort first applies an oblivious random permutation to the input elements, and then apply any (insecure) *comparison-based* sorting algorithm such as SPMS [30] to the permuted array. It is shown in [4] that this will give an oblivious sorting algorithm.

Our random permutation algorithm, Butterfly-Random-Permute (B-RPermute) builds on an algorithm for this problem given in Asharov et al. [4] to obtain improved bounds as well as cache-agnostic cache-efficiency. To attain an oblivious random permutation, a key step in [4] is to randomly assign elements to bins of $Z = \omega(\log n)$ capacity but without leaking the bin choices — we call this primitive *oblivious random bin assignment (ORBA)*. For simplicity, we shall assume $Z = \Theta(\log^2 n)$.

In Section 2.1 we describe our improvements to the ORBA algorithm in [4]. In Section 2.2 we describe Butterfly-Random-Permute and Butterfly-Sort.

2.1 Oblivious Random Bin Assignment: ORBA

We first give a brief overview of an ORBA algorithm by Asharov et al. [4]. By using an algorithm for parallel compaction in [5], this ORBA algorithm in [4] can be made to run with $O(n \log n)$ work and in $O(\log n \cdot \log\log n)$ parallel time on an EREW PRAM (though not on binary fork-join). In this ORBA algorithm [4], the n input elements are divided into $\beta = 2n/Z$ bins and each bin is then padded with $Z/2$ filler elements to a capacity of Z. We assume that β is a power of 2. Each real element in the input chooses a random label from the range $[0, \beta - 1]$ which expresses the desired bin. The elements are then routed over a 2-way butterfly network of $\Theta(\log n)$ depth, and in each layer of the butterfly network there are β bins, each of capacity Z. In every layer i of the network, input bins from the previous layer are paired up in an appropriate manner, and the real elements contained in the two input bins are obliviously distributed to two output bins in the next layer, based on the i-th bit of their random labels. For obliviousness, the output bins are also padded with filler elements to a capacity of Z.

Our 'meta-algorithm' Meta-ORBA saves an $O(\log\log n)$ factor in the PRAM running time while retaining the $O(n \log n)$ work. For this we use a $\gamma = \Theta(\log n)$-way butterfly network, with γ a power of 2, rather than 2-way. Therefore, in each layer of the butterfly network, groups of $\Theta(\log n)$ input bins from the previous layer are appropriately chosen, and the real elements in the $\log n$ input bins are obliviously distributed to $\Theta(\log n)$ output bins in the next layer, based on the next unconsumed $\Theta(\log\log n)$ bits in their random labels. Again, all bins are padded with filler elements to a capacity of Z so the adversary cannot tell the bin's actual load. To perform this $\Theta(\log n)$-way distribution obliviously, it suffices to invoke the AKS construction [2] $O(1)$ number of times.

We now analyze the performance of this algorithm.

- *For $O(1)$ invocations of AKS on $\Theta(\log n)$ bins each of size $Z = \Theta(\log^2 n)$:* Total work is $O(\log^3 n \cdot \log\log n)$ and the parallel time on an EREW PRAM is $O(\log\log n)$.
- *For computing a single layer, with $\beta/\Theta(\log n) = 2n/\Theta(\log^3 n)$ subproblems:* Total work for a single layer is $O(n \cdot \log\log n)$ and EREW PRAM time remains $O(\log\log n)$.
- The $\Theta(\log n)$-way butterfly network has $\Theta(\log n/\log\log n)$ layers, hence Meta-ORBA performs $O(n \log n)$ total work, and runs in $O(\log n)$ parallel time on an EREW PRAM.

Overflow analysis. Our Meta-ORBA algorithm has deterministic data access patterns that are independent of the input. However, if some bin overflows its capacity due to the random label assignment, the algorithm can lose real elements during the routing process. Asharov et al. [4] showed that for the special case $\gamma = 2$, the probability of overflow is upper bounded by $\exp(-\Omega(\log^2 n))$ assuming that the bin size $Z = \log^2 n$. Suppose now that $\gamma = 2^r$, then the elements in the level-i bins in our Meta-ORBA algorithm correspond exactly to the elements in the level-$(r \cdot i)$ bins in the ORBA algorithm in [4]. Since the failure probability $\exp(-\Omega(\log^2 n))$ is negligibly small in n, we have that for $Z = \log^2 n$ and $\gamma = \Theta(\log n)$, our Meta-ORBA algorithm obliviously realizes random bin assignment on an EREW PRAM with $O(n \log n)$ total work and $O(\log n)$ parallel runtime with negligible error.

We have now improved the parallel runtime by a $\log\log n$ factor relative to Asharov et al. [4] (even when compared against an improved version of their algorithm that uses new primitives such as parallel compaction [5]) but this is on an EREW PRAM. We will now translate this into Rec-ORBA, our cache-agnostic, binary fork-join implementation.

2.1.1 Rec-ORBA. We will assume a tall cache ($M = \Omega(B^2)$) for Rec-ORBA, our recursive cache-agnostic, binary fork-join implementation of our Meta-ORBA algorithm. We will also assume $M = \Omega(n^{1+\epsilon})$ for any given arbitrarily small constant $\epsilon > 0$. In the following we will use $\epsilon = 2$ for simplicity, i.e., $M = \Omega(\log^3 n)$.

In Rec-ORBA, we implement Meta-ORBA by recursively solving $\sqrt{\beta}$ subproblems, each with $\sqrt{\beta}$ bins: in each subproblem, we shall obliviously distribute the real elements in the $\sqrt{\beta}$ input bins into $\sqrt{\beta}$ output bins, using the $(1/2)\log\beta$ most significant bits (MSBs) in the labels. After this phase, we use a matrix transposition to bring the $\sqrt{\beta}$ bins with the same $(1/2) \cdot \log\beta$ MSBs together — these $\sqrt{\beta}$ bins now belong to the same subproblem for the next phase. where we recursively solve each subproblem defined above: for each subproblem, we distribute the $\sqrt{\beta}$ bins into $\sqrt{\beta}$ output bins based on the $(1/2) \cdot \log\beta$ least significant bits in the elements' random labels. Since the matrix transposition for Y bins each of capacity $Z = \Theta(\log^2 n)$ can be performed with $O(Y \cdot Z/B)$ cache misses, and $O(\log(Y \cdot Z)) = O(\log n)$ span, we have the following recurrences that characterize the cost of Rec-ORBA where Y denotes the current number of bins, and $Q(Y)$ and $T(Y)$ denote the cache complexity and span, respectively, to solve a subproblem containing Y bins:

$$Q(Y) = 2\sqrt{Y} \cdot Q(\sqrt{Y}) + O((Y \cdot \log^2 n)/B)$$
$$T(Y) = 2 \cdot T(\sqrt{Y}) + O(\log(Y \cdot \log^2 n))$$

The base conditions are as follows. Since $M = \Omega(\log^3 n)$ each individual $\Theta(\log n)$-way distribution instance fits in cache and incurs $O((1/B)\log^3 n)$ cache misses. Hence we have the base case $Q(Y) = O(Y \log^2 n/B)$ when $Y \log^2 n \leq M$. For the span, each individual $\Theta(\log n)$-way distribution instance works on $O(\log^3 n)$ elements and has $O(\log^2 \log n)$ span under binary forking, achieved by forking and joining $\log^3 n$ tasks at each level of the AKS sorting network.

Therefore, we have that for $\beta = 2n/Z$: $Q(\beta) = O((n/B)\log_M n)$ and $T(\beta) = O(\log n \cdot \log\log n)$. Finally, if we want $M = \Omega(\log^{1+\epsilon} n)$ rather than $M = \Omega(\log^3 n)$, we can simply parametrize the bin size to be $\log^{1+\epsilon/2} n$ and let $\gamma = \log^{\epsilon/2} n$. This gives rise to the following lemma.

Lemma 2.1. *Algorithm* Rec-ORBA *has cache-agnostic caching complexity $O((n/B)\log_M n)$ provided the tall cache has $M = \Omega(\log^{1+\epsilon} n)$, for any given positive constant ϵ. The algorithm performs $O(n \log n)$ work and has span $O(\log n \cdot \log\log n)$.*

2.2 Butterfly-Random-Permute and Butterfly-Sort

From ORBA's output (Meta-ORBA or Rec-ORBA) we obtain β bins where each bin contains real and filler elements. To obtain a random permutation of the input array, it is shown in [4, 19] that it suffices to do the following: Assign a $(\log n \cdot \log\log n)$-bit random label to each element, and sort the elements within each bin based their labels using bitonic sort, where all filler elements are treated as having the label ∞ and thus moved to the end of the bin. Finally, remove the filler elements from all bins, and output the result.

We will sort the elements in each bin by their labels with bitonic sort (with a cost of $O(\log\log n)$ for each comparison due to the large values for the labels). We have β parallel sorting problems on $\Theta(\log^2 n)$ elements, where each element has an $O(\log n \cdot \log\log n)$-bit label. Each sorting problem can be performed with $O(\log^4 \log n)$ span and $O(\log^2 n \cdot \log^3 \log n)$ work in binary fork-join using bitonic sort by paying a $O(\log\log n)$ factor for the work and span for each comparison due to the $O(\log n \cdot \log\log n)$-bit labels. The overall cost for this step across all subproblems is $O(n \cdot \log^3 \log n)$ work and the span remains $O(\log^4 \log n)$. Since each bitonic sort subproblem fits in cache, the caching cost is simply the scan bound $O(n/B)$. These bounds are dominated by our bounds for Rec-ORBA, so this gives an algorithm to generate a random permutation with the same bounds as Rec-ORBA.

Finally, once the elements have been permuted in random order, we can use any insecure comparison-based sorting algorithm to sort the permuted array. We use SPMS sort [30], the best cache-agnostic sorting algorithm for the binary fork-join model (CREW) to obtain Butterfly-Sort. Thus we achieve the bounds in Theorem 1.1 in the introduction, and we have a data-oblivious sorting algorithm that matches the performance bounds for SPMS [30], the current best insecure algorithm in the cache-agnostic binary fork-join model.

3 PRACTICAL AND EREW SORTING

So far, our algorithm relies on AKS [2] and the SPMS [30] algorithm as blackbox primitives and thus is not suitable for practical implementation. We now describe a potentially practical variant that is self-contained and implementable, and gets rid of both AKS and SPMS. To achieve this, we make two changes:

- *Bitonic Sort.* Our efficient method uses bitonic sort within it, so we first we give an efficient cache-agnostic, binary fork-join implementation of bitonic sort. A naïve binary fork-join implementation of bitonic sort would incur $O((n/B) \cdot \log^2 n)$ cache misses and $O(\log^3 n)$ span (achieved by forking and joining the tasks in each layer in the bitonic network). In our binary fork-join bitonic sort, the work remains $O(n \cdot \log^2 n)$ but the span reduces to $O(\log^2 n \cdot \log\log n)$ and the caching cost reduces to $O((n/B) \cdot \log_M n \cdot \log(n/M))$. (See Section 3.1.)
- *Butterfly Random Sort.* Second, we present a simple algorithm to sort a randomly permuted input. This algorithm Butterfly-Random-Sort (or B-RSort) *uses the same structure as* B-RPermute but uses a sorted set of pivots to determine the binning of the elements. Outside of the need to initially sort $O(n/\log n)$ random pivot elements, this algorithm has the same performance bounds as B-RPermute. (See Section 3.2.)

We obtain a simple sorting algorithm, which we call Butterfly-Butterfly-Sort or BB-Sort by running B-RPermute followed by B-RSort on the input: The first call outputs a random permutation of the input and the second call sorts this sequence since B-RSort correctly sorts a randomly permuted input. We highlight two implementation of BB-Sort:

Practical BB-Sort. By using our improved bitonic sort algorithm in place of AKS networks for small subproblems in BB-Sort (both within B-RPermute and B-RSort), we obtain a simple and practical data-oblivious sorting algorithm that has optimal cache-agnostic cache complexity if $M = \Omega(\log^{2+\epsilon} n)$ and incurs only an $O(\log\log n)$ blow-up in work and an $O(\log n)$ blow-up in span. This algorithm has small constant factors, with each use of bitonic sort contributing a constant factor of 1/2 to the bounds for the number of comparisons made.

EREW BB-Sort. For a more efficient EREW binary fork-join sorting algorithm, we retain the AKS network in the butterfly computations in both B-RPermute and B-RSort. We continue to sort the pivots at the start of B-RSort with bitonic sort. Thus the cost of the overall algorithm is dominated by the cost to sort the $n/\log n$ pivots using bitonic sort, and we obtain a simple self-contained EREW binary fork-join sorting algorithm with optimal work and cache-complexity and $\tilde{O}(\log^2 n)$ span.

Theorem 3.1 states the bounds on these two variants. In the rest of this section we briefly discuss bitonic sort in Section 3.1 and then we present and analyze B-RSort in Section 3.2.

Theorem 3.1. *For $M = \Omega(\log^{2+\epsilon} n)$:*
(i) Practical BB-Sort runs in $O(n \log n \log\log n)$ work, cache-agnostic caching cost $Q_{sort}(n)$, and $O(\log^2 n \cdot \log\log n)$ span.
(ii) EREW BB-Sort runs in $O(W_{sort}(n))$ work, cache-agnostic caching cost $O(Q_{sort}(n))$, and $O(\log^2 n \cdot \log\log n)$ span.
Both algorithms are data oblivious with negligible error.

Table 3 lists the bounds for our sorting and permuting algorithms.

3.1 Binary Fork-Join Bitonic Sort

Bitonic sort has $\log n$ stages of Bitonic-Merge. We observe that Bitonic-Merge is a butterfly network and hence it has a binary fork-join algorithm with $O(n \log n)$ work, $O(Q_{sort}(n))$ caching cost, and $O(\log n \log\log n)$ span. Using this within the bitonic sort algorithm we obtain the following theorem. (Details are in the full write-up [50].)

Theorem 3.2. *There is a data oblivious cache-agnostic binary fork-join implementation of bitonic sort that can sort n elements in $O(n \log^2 n)$ work, $O(\log^2 n \cdot \log\log n)$ span, and $O((n/B) \cdot \log_M n \cdot \log(n/M))$ cache complexity, when $n > M \geq B^2$.*

3.2 Sorting a Random Permutation: Butterfly-Random-Sort (B-RSort)

As mentioned, Butterfly-Random-Sort (B-RSort) uses the same network structure as Rec-ORBA. During pre-processing, we pick and sort a random set of pivot elements. Initially, elements in the input array **I** are partitioned in bins containing $\Theta(\log^3 n)$ elements each. Then, the elements traverse a γ-way butterfly network where $\gamma = \Theta(\log n)$ and is chosen to be a power of 2: at each step, we use

Table 3: Permute & Sort. $\widetilde{O}$ hides a single $\log\log n$ factor. All algorithms except B-RSort are data-oblivious and all have negligible error, except Bitonic-Sort, which is deterministic. Optimal bounds for general sorting (after the first two rows) are displayed in a box.

Algorithm	Function	work	span	cache
B-RPermute	Outputs random permutation of input array	$O(n \log n)$	$\widetilde{O}(\log n)$	$O(\frac{n}{B} \log_M n)$
B-RSort	Sorts a randomly permuted input array	$O(n \log n)$	$\widetilde{O}(\log n)$	$O(\frac{n}{B} \log_M n)$
Butterfly-Sort	Sorts input array using B-RPermute and SPMS	$\boxed{O(n \log n)}$	$\widetilde{O}(\log n)$	$\boxed{O(\frac{n}{B} \log_M n)}$
Fork-join Bitonic-Sort	Based on deterministic sorting network	$O(n \log^2 n)$	$\widetilde{O}(\log^2 n)$	$O(\frac{n}{B} \log_M n \cdot \log \frac{n}{M})$
BB-Sort	Sorts using B-RPermute and B-RSort	Practical: $\widetilde{O}(n \log n)$ EREW: $\boxed{O(n \log n)}$	$\widetilde{O}(\log^2 n)$ $\widetilde{O}(\log^2 n)$	$\boxed{O(\frac{n}{B} \log_M n)}$ $\boxed{O(\frac{n}{B} \log_M n)}$

sorting to distribute a collection of γ bins at the previous level into γ bins at the next level. Instead of determining the output bin at each level by the random label assigned to an element as in Rec-ORBA, here each bin has a range determined by two pivots, and each element will be placed in the bin whose range contains the element's value. Also, in Rec-ORBA we needed to use filler elements to hide the actual load of the intermediate bins. B-RSort does not need to be data-oblivious since its input is a random permutation of the original input, hence we *do not need filler elements* in B-RSort.

Pivot selection. The pivots are chosen in a pre-processing phase, so that we can guarantee that every bin has $O(\log^3 n)$ elements with all but negligible failure probability, as follows.

(1) First, generate a sample Π of size close to $n/\log n$ from the input array $\mathbf{I}$ by sampling each element with probability $1/\log n$. We then sort Π using bitonic sort.
(2) In the sorted version of Π, every element whose index is a multiple of $\log^2 n$ is moved into a new sorted array pivots. Pad the pivots array with an appropriate number of ∞ pivots such that its length plus 1 would be a power of 2.

Using a Chernoff bound it is readily seen that the size of Π is $(n/\log n) \pm n^{3/4}$ except with negligible probability. We choose every $(\log^2 n)$-th element after sorting Π to form our set of pivots. Hence r, the number of pivots, is $\frac{n}{\log^3 n} + o(n/\log^3 n)$ except with negligible probability.

By Theorem 3.2, sorting $\Theta(n/\log n)$ samples incurs $O(n \log n)$ total work, $O((n/B) \log_M n)$ cache complexity, and span $O(\log^2 n \log\log n)$ — this step will dominate the span of our overall algorithm. The second step of the above algorithm can be performed using prefix-sum, incurring total work $O(n)$, cache complexity $O(n/B)$, and span $O(\log n)$.

Sort into bins. Suppose that $r - 1$ pivots were selected above; the pivots define a way to partition the values being sorted into r roughly evenly loaded *regions*, where the i-th region includes all values in the range $(\text{pivots}[i-1], \text{pivots}[i]]$ for $i \in [r]$. For convenience, we assume $\text{pivots}[0] = -\infty$ and $\text{pivots}[r] = \infty$. We first describe algorithm Rec-SBA (Recursive Sort Bin Assignment), which partially sorts the randomly permuted input into a sequence of bins, and then we sort within each bin to obtain the final sorted output. This is similar to applying Rec-ORBA followed by sorting within bins in B-RPermute.

In algorithm Rec-SBA the input array $\mathbf{I}$ will be viewed as $r = \Theta(n/\log^3 n)$ bins each containing $n/r = \Theta(\log^3 n)$ elements. The Rec-SBA algorithm first partitions the input bins into $\sqrt{r}$ groups each containing $\sqrt{r}$ consecutive bins. Then, it recursively sorts each group using appropriate pivots among the precomputed pivots. At this point, it applies a matrix transposition on the resulting bins. After the matrix transposition all elements in each group of $\sqrt{r}$ consecutive bins will have values in a range between a pair of pivots $\sqrt{r}$ apart in the sorted list of pivots, and this group is in its final sorted position relative to the other groups. Now, for a second time, the algorithm recursively sorts each group of consecutive $\sqrt{r}$ bins, using the appropriate pivots and this will place each element in its final bin in sorted order. Our algorithm also guarantees that the pivots are accessed in a cache-efficient manner.

We give a more formal description in Rec-SBA$^{\gamma}(\mathbf{I}, \text{pivots})$. Recall that $\gamma = \Theta(\log n)$ is the branching factor in the butterfly network.

Final touch. To output the fully sorted sequence, BB-Sort simply applies bitonic sort to each bin in the output of Rec-SBA and outputs the concatenated outcome. The cost of this step is asymptotically absorbed by Rec-SBA in all metrics.

Overflow analysis. It remains to show that no bin will receive more than $C \log^3 n$ elements for some suitable constant $C > 1$ except with negligible probability.

Let us use the term Meta-SBA to denote the non-recursive meta-algorithm for Rec-SBA. For overflow analysis, we can equivalently analyze Meta-SBA. Consider a collection of $\gamma = \Theta(\log n)$ bins in the j-th subproblem in level $i-1$ of Meta-SBA whose contents are input to a bitonic sorter. Let us refer to this as group $(i-1, j)$. These elements will be distributed into γ bins in level i using the $\gamma - 1$ pivots with label pair $(i-1, j)$. The elements in these γ bins in group $(i-1, j)$ came from γ^{i-1} bins in the first level, each containing exactly $\log^2 n$ elements from input array $\mathbf{I}$. Let U be the set of elements in these γ^{i-1} bins in the first level, so size of U is $u = \gamma^{i-1} \cdot n/r = \gamma^{i-1} \cdot \Theta(\log^3 n)$.

Let us fix our attention on a bin b in the i-th level into which some of the elements in group $(i-1, j)$ are distributed after the bitonic sort. Consider the pair of pivots p, q that are used to determine the contents of bin b (we allow $p = -\infty$ and $q = \infty$ to account for the first and last segment). This pair of pivots is used to split the elements in the level $i-1$ group $(i-1, j)$ hence the pivots p and q are r/γ^{i-1} apart in the sorted sequence $\text{pivots}[1..r]$. We also

know that the number of elements in **I** with ranks between any two successive pivots in pivots$[1..r]$ is at most $\log^3 n + o(\log^3 n)$ except with negligible probability. Hence the number of elements in the input array **I** that have ranks between the ranks of these two pivots is $k = (r/\gamma^{i-1}) \cdot \log^3 n$. Recalling that $r \le 2n/\log^3 n$ (except with negligible probability), we have $k = (r/\gamma^{i-1}) \cdot \log^3 n \le 2n/(\gamma^{i-1})$ except with negligible probability.

Let Y_b be the number of elements in bin b. These are the elements in U that have rank between the ranks of p and q. The random variable Y_b is binomially distributed on $u = |U|$ elements with success probability at most $k/n = 2/\gamma^{i-1}$. Hence, $\mathbb{E}[Y_b] \le u \cdot k/n = \Theta(\log^3 n)$ and using Chernoff bounds, $Y_b < \Theta(\log^3 n) + o(\log^3 n)$ with all but negligible probability.

Rec-SBA$^\gamma$(**I**, pivots)

Input: The input array **I** contains β bins where β is assumed to be a power of 2. The array pivots contains $\beta - 1$ number of pivots that define β number of regions, where the i-th region is (pivots$[i-1]$, pivots$[i]$]. Each element in **I** will go to an output bin depending on which of the β regions its value falls into.

Algorithm: If $\beta \le \gamma$, use bitonic sort to assign the input array **I** to a total of γ bins based on pivots. Return the resulting list of β bins.

Else, proceed with the following recursive algorithm.

(1) Divide the input array **I** into $\sqrt{\beta}$ partitions where each partition contains exactly $\sqrt{\beta}$ consecutive bins[a] Henceforth let $\mathbf{I}^j$ denote the j-th partition.

(2) Let pivots$'$ be constructed by taking every pivot in pivots whose index is a multiple of $\sqrt{\beta}$.
In parallel: For each $j \in [\sqrt{\beta}]$, let $\mathsf{Bin}^j_1, \ldots, \mathsf{Bin}^j_{\sqrt{\beta}} \leftarrow$ Rec-SBA$^\gamma(I^j$, pivots$')$.

(3) Let Bins be a $\sqrt{\beta} \times \sqrt{\beta}$ matrix where the j-th row is the list of bins $\mathsf{Bin}^j_1, \ldots, \mathsf{Bin}^j_{\sqrt{\beta}}$. Note that each element in the matrix is a bin. Now, perform a matrix transposition: TBins $\leftarrow$ Bins^T.
Henceforth, let TBins$[i]$ denote the i-th row of TBins consisting of $\sqrt{\beta}$ bins,

(4) In parallel: For $i \in [\sqrt{\beta}]$:
let $\widetilde{\mathsf{Bin}}^i_1, \ldots, \widetilde{\mathsf{Bin}}^i_{\sqrt{\beta}} \leftarrow$ Rec-SBA$^\gamma$(TBins$[i]$, pivots$[(i-1)\sqrt{\beta}+1..i\cdot\sqrt{\beta}-1]$)

(5) Return the concatenation $\widetilde{\mathsf{Bin}}^1_1, \ldots, \widetilde{\mathsf{Bin}}^1_{\sqrt{\beta}}, \widetilde{\mathsf{Bin}}^2_1, \ldots, \widetilde{\mathsf{Bin}}^2_{\sqrt{\beta}}, \ldots, \ldots, \widetilde{\mathsf{Bin}}^{\sqrt{\beta}}_1, \ldots, \widetilde{\mathsf{Bin}}^{\sqrt{\beta}}_{\sqrt{\beta}}$.

[a]For simplicity, we assume that β is a perfect square in every recursive call.

Performance analysis. By the above overflow analysis we can assume that if any bin in Rec-SBA receives more than $C\log^3 n$ elements for some suitable constant $C > 1$, the algorithm fails, since we have shown above that this happens only with negligibly small in n probability.

For the practical version, we replace the AKS with bitonic on problems of size $\Theta(\log^3 n)$, the total work becomes $O(n \log n \log\log n)$, and the span becomes $O(\log n \log\log n \log\log\log n)$.

To see the cache complexity, observe that Line 2 requires a sequential scan through the array pivots that is passed to the current recursive call, writing down $\beta - 1$ number of pivots, and making $\sqrt{\beta}$ copies of them to pass one to each of the $\sqrt{\beta}$ subproblems. Now, Line 4 divides pivots into $\sqrt{\beta}$ equally sized partitions, removes the rightmost boundary point of each partition, passes each partition (now containing $\sqrt{\beta} - 1$ pivots) to one subproblem. Both Lines 2 and 4 are upper bounded by the scan bound $Q_{\text{scan}}(\beta \cdot \Theta(\log^3 n))$, and so is the matrix transposition in Line 3. Therefore, Rec-SBA achieves optimal cache complexity similar to Rec-ORBA, but now assuming that $M = \Omega(\log^4 n)$.

Our practical variant. Putting everything all together, in our practical variant, we use RB-Permute (instantiated with bitonic sort for each poly-logarithmically sized problem), and similarly in Rec-SBA and in the final sorting of the bins. The entire algorithm — outside of sorting the $O(n/\log n)$ pivots — incurs $O(n \log n \log\log n)$ total work, $O(\log n \log\log n \log\log\log n)$ span, and optimal cache complexity $O((n/B)\log_M n)$. When we add the cost of sorting the pivots, the work and caching bounds remain the same but the span becomes $O(\log^2 n \cdot \log\log n)$. The constants hidden in the big-O are small: for total work, the constant is roughly 18; for cache complexity, the constant is roughly 6; and for span, the constant is roughly 1.

Improving the requirement on M. We can improve the constraint on M to $M = \Omega(\log^{2+\epsilon} n)$ for an arbitrarily small $\epsilon \in (0, 1)$, if we make the following small modifications:

(1) Choose every $(\log n)^{1+\epsilon}$-th element from the samples as a pivot; and let $\gamma = \Theta(\log n)$. Note that in this case, the total number of pivots is $r = \Theta(n/\log^{2+\epsilon} n)$.

(2) Earlier, we divided the input into r bins each with n/r elements; but now, we divide the input array into $r \cdot \gamma$ bins, each filled with $n/(r\cdot\gamma) = \Theta(\log^{1+\epsilon} n)$ elements. Because there are γ times more bins than pivots now, in the last level of the meta-algorithm, we will no longer have any pivots to consume – but this does not matter since we can simply sort each group of γ bins in the last level.

Finally, for the EREW version, we retain the AKS networks for the *polylogn*-size subproblems, so the only step that causes an increase in performance bounds is the use of bitonic sort to sort $n/\log n$ pivots. Using our EREW binary fork-join algorithm for this step, we achieve the bounds stated in Theorem 3.1.

4 OBLIVIOUS, BINARY FORK-JOIN SIMULATION OF PRAMS

We show that our new oblivious sorting algorithm gives rise to oblivious simulation of CRCW PRAMs in the binary fork-join model with non-trivial efficiency guarantees. In Section 4.1 we give an oblivious simulation of PRAM in the binary fork-join model that is only efficient if the space s consumed by the original CRCW PRAM is small (e.g., roughly comparable to the number of cores p). Then in Section 4.2, we present another simulation that yields better efficiency when the original PRAM may consume large space.

4.1 Oblivious, Binary Fork-Join Simulation of Space-Bounded PRAMs

We give an oblivious simulation of PRAM in the binary fork-join model that is only efficient if the space s consumed by the original CRCW PRAM is small (e.g., roughly comparable to the number of cores p). Our oblivious simulation is based on a sequential cache-efficient emulation of a PRAM step given in [21], which in turn is based on a well-known emulation of a p-processor CRCW PRAM on an EREW PRAM in $O(\log p)$ parallel time with p processors (see, e.g., [43]). To ensure data-obliviousness, we will make use of the following well-known oblivious building block, *send-receive*:[3].

send-receive: The input has a source array and a destination array. The source array represents n senders, each of whom holding a key and a value; it is promised that all keys are distinct. The destination array represents n' receivers each holding a key. Each receiver needs to learn the value corresponding to the key it is requesting from one of the sources. If the key is not found, the receiver should receive $\perp$. (Note that although each receiver wants only one value, a sender can send its values to multiple receivers.)

The full paper [50] explains how to accomplish oblivious send-receive within the sorting bound in the binary fork-join model. Using this primitive, the PRAM simulation on binary fork-join works as follows. We separate each PRAM step into a read step, followed by a local computation step (for which no simulation is needed), followed by a write step. Suppose that in some step of the original CRCW PRAM, each of the p processors has a request of the form $(\text{read}, \text{addr}_i, i)$ or $(\text{write}, \text{addr}_i, v_i, i)$ where $i \in [p]$, indicates that either it wants to read from logical address addr_i or it wants to write v_i to addr_i.

(1) For a read step, using oblivious send-receive, every request obtains a fetched value from the memory array.
(2) In a write step we need to perform the writes to the memory array. To do this, we first perform a conflict resolution step in which we suppress the duplicate writes to the same address in the incoming batch of p requests. Moreover, if multiple processors want to write to the same address, the one that is defined by the original CRCW PRAM's priority rule is preserved; and all other writes to the same address are replaced with fillers. This can be accomplished with $O(1)$ oblivious sorts. Now, with oblivious send-receive, every address in the memory array can be updated with its new value.

Thus we have the following theorem.

Theorem 4.1 (Space-bounded PRAM-on-OBFJ). *Let $M > \log^{1+\epsilon} s$ and $s \geq M \geq B^2$. Any p-processor CRCW PRAM algorithm that uses space at most s can be converted to a functionally equivalent, oblivious and cache-agnostic algorithm in the binary fork-join model, where each parallel step in the CRCW can be simulated with $O(W_{\text{sort}}(p+s))$ work, $O(Q_{\text{sort}}(p+s))$ cache complexity, and $O(T_{\text{sort}}(p+s))$ span.*

Using the above theorem, any fast and efficient PRAM algorithm that uses linear space (and many of them do) will give rise to an oblivious algorithm in the cache-agnostic binary fork-join model with good performance. In Section 5 we will use this theorem to obtain new results for graph problems such as tree contraction, graph connectivity, and minimum spanning forest — for all three problems, the performance bounds of our new data-oblivious algorithms improve on the previous best results even for algorithms that need not be data oblivious.

[3]The send-receive abstraction is often referred to as oblivious routing in the data-oblivious algorithms literature [15, 17, 18]. We avoid the name "routing" because of its other connotations in the algorithms literature.

4.2 Oblivious, Binary Fork-Join Simulation of Large-Space PRAMs

In this section, we describe a simulation strategy that achieves better bounds when the PRAM's space can be large. This is obtained by combining our Butterfly Sort with the prior work of Chan et al. [18]. We assume familiarity with [18] here; the full paper [50] has an overview of this prior result.

To efficiently simulate any CRCW PRAM as a data-oblivious, binary fork-join program, we make the following modifications to the algorithm in [18]. First, we switch several core primitives they use to our cache-agnostic, binary fork-join implementations: we replace their oblivious sorting algorithm with our new sorting algorithm in the cache-agnostic, binary fork-join model; and we replace their oblivious "send-receive", "propagation", and "aggregation" primitives with our new counterparts in the cache-agnostic, binary fork-join model (see Section 5).

Second, we make the following modifications to improve the cache complexity:

- We store all the binary-tree data structures (called ORAM trees) in Chan et al. [18] in an Emde Boas layout. In this way, accessing a tree path of length $O(\log s)$ incurs only $O(\log_B s)$ cache misses.
- The second modification is in the "simultaneous removal of visited elements" step in the maintain phase of the algorithm in [18]. In this subroutine, for each of $O(\log s)$ recursion levels: each of the p processors populates a column of a $(\log s) \times p$ matrix, and then oblivious aggregation is performed on each row of the matrix. To make this cache efficient, we can have each of the p threads populate a row of a $p \times (\log s)$ matrix, and then perform matrix transposition. Then we then apply oblivious aggregation to each row of the transposed matrix.

Plugging in these modifications to Chan et al. [18] we obtain the following theorem:

Theorem 4.2 (PRAM-on-OBFJ: Simulation of CRCW PRAM on Oblivious, binary fork-join). *Suppose that $M > \log^{1+\epsilon} s$, $s \geq M \geq B^2$, and $s \geq p$. Any p-processor CRCW PRAM using at most s space can be converted to a functionally equivalent, oblivious program in the cache-agnostic, binary fork-join model, where each parallel step in the CRCW can be simulated with $O(p \log^2 s)$ total work, $O(\log s \cdot \log\log s)$ span, and $O(\log s \cdot ((p/B) \cdot \log_M p + p \cdot \log_B s))$ cache complexity.*

Proof. The total work is the same as Chan et al. [18] since none of our modifications incur asymptotically more work.

Our span matches the PRAM depth of Chan et al. [18], that is, $O(\log s \log\log s)$, because all of our primitives, including sorting, aggregation, propagation, and send-receive incur at most $O(\log s \log\log s)$ span. The bottleneck in PRAM depth of Chan et al. [18] comes not from the sorting/aggregation/propagation/send-receive, but from having to *sequentially* visit $O(\log s)$ recursion levels during the fetch

phase, where for each recursion level, we need to look at a path in an ORAM tree of length $O(\log s)$, and find the element requested along this path — this operation takes $O(\log \log s)$ PRAM depth as well as $O(\log \log s)$ span in a binary fork-join model. Finally, the new matrix transposition modification in the "simultaneous removal" step does not additionally increase the span.

For cache complexity, there are two parts, the part that comes from $\log s$ number of oblivious sorts on $O(p)$ number of elements — this incurs $O((p/B) \cdot \log_M p \cdot \log s)$ cache misses in total. The second part comes from having to access $p \cdot \log s$ ORAM tree paths in total where each path is of length $O(\log s)$. If all the ORAM trees are stored in Emde Boas layout, this part incurs $O(p \cdot \log s \cdot \log_B s)$ cache misses. These two parts dominate the cache complexity, and the caching cost of all other operations is absorbed by the sum of these two costs. □

This result can be viewed as a generalization of the prior best Oblivious Parallel RAM (OPRAM) result [17, 18]. The prior result [17, 18] shows that any p-processor CRCW PRAM can be simulated with an oblivious CREW PRAM where each CRCW PRAM step is simulated with $p \log^2 s$ total work and $O(\log s \log \log s)$ parallel runtime (without the binary-forking requirement). Essentially our result matches the best known result, and we further show that the extra binary-forking requirement does not incur additional overhead during this simulation. We additionally achieve cache-agnostic cache efficiency: The prior work [17, 18] also did not explicitly consider cache complexity.

5 APPLICATIONS

We sketch our results for various applications of our new sorting algorithm, assuming knowledge of results for the non-oblivious case. (Self-contained descriptions of the algorithms are in the full paper [50].) All of our data-oblivious algorithms either match or outperform the best known bounds for their insecure counterparts (in the cache-agnostic, binary fork-join model). Euler tour and tree contraction were also considered in data-oblivious algorithms [7, 39, 40] but the earlier works are inherently sequential.

Basic data-oblivious primitives. *Aggregration and propagation in a sorted array* were primitives used in the simulation algorithms in Section 4.2. These can be readily formulated as segmented prefix sums and can be computed data obliviously within the scan bound ($O(\log n)$ span, $O(n)$ work and $O(n/B)$ cache-agnostic cache misses for an n-array.) The third primitive used in Section 4.2, *send-receive*, can be performed obliviously with a constant number of sorts and scans and hence achieves the sort bound. (See the full paper [50] for details.)

List Ranking and Applications. To realize list ranking obliviously, we first apply Rec-OPerm to randomly permute the input array and then apply a non-oblivious list ranking algorithm [26] to match the bounds for the insecure case: $O(W_{sort}(n))$, $O(Q_{sort}(n))$, and $O(T_{sort}(n) \cdot \log n)$.

For Euler tour in a unrooted tree we use the insecure algorithm [26, 43] that reduces the problem to list ranking. In this reduction there is an insecure step where each edge (u, v) needs to locate the successor of edge (v, u) in v's circular adjacency list. This can be seen as an instance of send-receive and hence can be performed obliviously within the sort bound, so we can compute the Euler tour obliviously with the same bounds as list ranking. Once we have an Euler we can compute common tree functions (e.g., pre and post order numbering and depth of each node) for any given root with list ranking. Thus we obtain the bounds stated in part (i) of Theorem 5.1 .

Results Using PRAM Simulation. Using our PRAM simulation results in Section 4.2 we obtain oblivious algorithms with improved bounds for over what is generally claimed for the insecure algorithms. Our data-oblivious algorithms are randomized due to the use of our randomized sorting algorithm; for the prior best insecure algorithms it suffices to use the SPMS sorting algorithm [30] and therefore, they are deterministic (but not data oblivious).

Tree contraction. The EREW PRAM tree contraction algorithm of Kosaraju and Delcher [44] runs in $\log n$ phases where the i-th phase, $i \geq 1$, performs a constant number of parallel steps with $O(n/c^{i-1})$ work on $O(n/c^{i-1})$ data items, for a constant $c > 1$. Further, the rate of decrease is fixed and data independent, and at the end of each phase, every memory location knows whether it is still needed in future computation. Since the memory used is geometrically decreasing in successive phases, a constant fraction of the memory locations will be no longer needed at the end of each phase.

We now apply our earlier PRAM simulation result in Theorem 4.1 in a slightly non-blackbox fashion: We simulate each PRAM phase, always using the actual number of processors needed in that step, which is the work for that step in the work-time formulation. Additionally, we introduce the following modification: at the end of each phase, we use oblivious sort move the memory entries no longer needed to the end, effectively removing them from the future computation. This gives the first result in in Theorem 5.1 below.

Connected components and minimum spanning forest. These two problems can be solved on the PRAM on an n-node, m-edge graph in $T(n) = O(\log n)$ parallel time and with $O(m+n)$ space and number of processors [49, 53]. Hence using Theorem 4.1 for each of the $T(n)$ steps we achieve the bounds stated in part (ii) of the theorem below. This improves over the generally cited bounds for the insecure case, which are obtained through explicit algorithms and have span that is large (i.e. slower) by a $\log n$ factor.

Theorem 5.1. *Suppose that $M > \log^{1+\epsilon}(m+n)$ and $m+n \geq M \geq B^2$. Then, we have the following bounds for oblivious cache-agnostic, binary fork-join algorithms.*

(i) List ranking, Euler tour and tree functions, and tree contraction in $O(W_{\text{sort}}(n))$ work, $O(Q_{\text{sort}}(n))$ cache complexity, and $O(\log n \cdot T_{\text{sort}}(n))$ span, where n is the size of the input.

(ii) Connected components and minimum spanning forest in a graph with n nodes and m edges in $O(\log n \cdot W_{\text{sort}}(m+n))$ work, $O(\log n \cdot Q_{\text{sort}}(m+n))$ cache complexity, and $O(\log n \cdot T_{\text{sort}}(m+n))$ span.

REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Bar Harbor, Maine, USA). Association for Computing Machinery, 1–12.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An O(N Log N) Sorting Network. In *STOC*.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Puerto Vallarta, Mexico). Association for Computing Machinery, New York, NY, USA, 119–129.

[4] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *3rd Symposium on Simplicity in Algorithms, SOSA@SODA*, Martin Farach-Colton and Inge Li Gørtz (Eds.). SIAM, 8–14.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2020. Oblivious Parallel Tight Compaction. In *Information-Theoretic Cryptography (ITC)*.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2020. Optimal Oblivious Parallel RAM. Manuscript in preparation. Personal communication with Asharov et al.

[7] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ASIA CCS*.

[8] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (San Francisco, California). Society for Industrial and Applied Mathematics, USA, 501–510.

[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 355–366. https://doi.org/10.1145/1989493.1989553

[10] Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively Sharing a Cache among Threads. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 235–244. https://doi.org/10.1145/1007912.1007948

[11] 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), 281–321. https://doi.org/10.1145/301970.301974

[12] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low Depth Cache-Oblivious Algorithms. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Thira, Santorini, Greece). Association for Computing Machinery, New York, NY, USA, 189–199.

[13] Robert D. Blumofe and Charles E. Leiserson. 1993. Space-Efficient Scheduling of Multithreaded Computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC)* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 362–371.

[14] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[15] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2015. Oblivious Parallel RAM. In *Theory of Cryptography Conference (TCC)*.

[16] Zoran Budimlić, Vincent Cavé, Raghavan Raman, Jun Shirako, Sağnak Taşundefinedrlar, Jisheng Zhao, and Vivek Sarkar. 2011. The Design and Implementation of the Habanero-Java Parallel Programming Language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 185–186. https://doi.org/10.1145/2048147.2048198

[17] T.-H. Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs. In *Theory of Cryptography Conference, (TCC)*.

[18] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. 2017. On the Depth of Oblivious Parallel RAM. In *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, 567–597.

[19] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2018. Cache-Oblivious and Data-Oblivious Sorting and Applications. In *SODA*.

[20] 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 *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 519–538.

[21] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1995. External-Memory Graph Algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '95)*. Society for Industrial and Applied Mathematics, USA, 139–149.

[22] Rezaul A. Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. ACM SPAA*.

[23] Rezaul A. Chowdhury and Vijaya Ramachandran. 2010. The Cache-oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation. *Theory of Computing Systems* 47, 1 (2010), 878–919. Preliminary version in SPAA 2007.

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

[25] Richard Cole and Vijaya Ramachandran. 2010. Resource Oblivious Sorting on Multicores. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 6198)*, Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis (Eds.). Springer, 226–237.

[26] Richard Cole and Vijaya Ramachandran. 2012. Efficient Resource Oblivious Algorithms for Multicores with False Sharing. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 201–214.

[27] Richard Cole and Vijaya Ramachandran. 2012. Revisiting the cache miss analysis of multithreaded algorithms. In *Proc. LATIN*.

[28] Richard Cole and Vijaya Ramachandran. 2013. Analysis of randomized work-stealing with false sharing. In *Proc. IPDPS*.

[29] Richard Cole and Vijaya Ramachandran. 2017. Bounding Cache Miss Costs of Multithreaded Computations Under General Schedulers. In *Proc. ACM SPAA*.

[30] Richard Cole and Vijaya Ramachandran. 2017. Resource Oblivious Sorting on Multicores. *ACM Trans. Parallel Comput.* 3, 4, Article 23 (March 2017), 31 pages. Preliminary version in ICALP 2010.

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

[32] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 285–297.

[33] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)* (Montreal, Quebec, Canada). Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[34] M. Frigo and V. Strumpen. 2009. The cache complexity of multithreaded cache-oblivious algorithms. *Theory of Computing Systems* 45 (2009), 203–233.

[35] Github. [n.d.]. https://github.com/oneapi-src/oneTBB.

[36] O. Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*.

[37] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

[38] Michael T. Goodrich. 2014. Zig-zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in O(N Log N) Time. In *STOC*.

[39] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. 2013. Graph Drawing in the Cloud: Privately Visualizing Relational Data Using Small Working Storage. In *Graph Drawing*, Walter Didimo and Maurizio Patrignani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 43–54.

[40] Michael T. Goodrich and Joseph A. Simons. 2014. Data-Oblivious Graph Algorithms in Outsourced External Memory. In *Combinatorial Optimization and Applications - 8th International Conference, COCOA 2014, Wailea, Maui, HI, USA, December 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8881)*. Springer, 241–257.

[41] Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium (NDSS)*.

[42] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. 2020. Optimal Oblivious Priority Queues and Offline Oblivious RAM. In *ACM-SIAM SODA*.

[43] Richard M. Karp and Vijaya Ramachandran. 1990. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 869–942.

[44] S. Rao Kosaraju and Arthur L. Delcher. 1988. Optimal Parallel Evaluation of Tree-Structured Computations by Raking. In *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings (Lecture Notes in Computer Science, Vol. 319)*, John H. Reif (Ed.). Springer, 101–110.

[45] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, USA, 359–376.

[46] Microsoft. [n.d.]. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl?redirectedfrom=MSDN.

[47] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 377–394.

[48] Oracle. [n.d.]. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[49] Seth Pettie and Vijaya Ramachandran. 2002. A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest. *SIAM J. Comput.* 31, 6 (2002), 1879–1895.

[50] Vijaya Ramachandran and Elaine Shi. 2020. Data Oblivious Algorithms for Multicores. arXiv:2008.00332 [cs.DC]
[51] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 842–858.
[52] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*.
[53] Yossi Shiloach and Uzi Vishkin. 1982. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms* 3, 1 (1982), 57–67.
[54] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, USA, 640–656.