Parallel Algorithms for Asymmetric Read-Write Costs

Naama Ben-David† Guy E. Blelloch† Jeremy T. Fineman‡ Phillip B. Gibbons†
Yan Gu† Charles McGuffey† Julian Shun*

†Carnegie Mellon University ‡Georgetown University *UC Berkeley

{nbendavi,guyb,gibbons,yan.gu,cmcguffe}@cs.cmu.edu
jfineman@cs.georgetown.edu jshun@eecs.berkeley.edu

ABSTRACT

Motivated by the significantly higher cost of writing than reading in emerging memory technologies, we consider parallel algorithm design under such asymmetric read-write costs, with the goal of reducing the number of writes while preserving work-efficiency and low span. We present a nested-parallel model of computation that combines (i) small per-task stack-allocated memories with symmetric read-write costs and (ii) an unbounded heap-allocated shared memory with asymmetric read-write costs, and show how the costs in the model map efficiently onto a more concrete machine model under a work-stealing scheduler. We use the new model to design reduced-write, work-efficient, low-span parallel algorithms for a number of fundamental problems such as reduce, list contraction, tree contraction, breadth-first search, ordered filter, and planar convex hull. For the latter two problems, our algorithms are output-sensitive in that the work and number of writes decrease with the output size. We also present a reduced-write, low-span minimum spanning tree algorithm that is nearly work-efficient (off by the inverse Ackermann function). Our algorithms reveal several interesting techniques for significantly reducing shared memory writes in parallel algorithms without asymptotically increasing the number of shared memory reads.

1. INTRODUCTION

We are on the cusp of the emergence of a new wave of nonvolatile memory technologies that are projected to become the dominant type of main memory in the near future [1, 2, 40, 54]. A key property of these new memory technologies (e.g., phase-change memory, spin-torque transfer magnetic RAM, and memristor-based resistive RAM) is their asymmetric read-write costs: Writes can be an order of magnitude or more higher energy, higher latency, and lower (per-module) bandwidth than reads [3, 8, 11, 12, 15, 22, 23, 32, 33, 36, 46, 52]. This high cost for writes motivates the design of models that reflect this asymmetry and "write-efficient" algorithms that perform well under such models by reducing their number of writes.

Prior work has studied read-write asymmetries in several contexts. Work targetting NAND Flash memory [9, 24, 25, 44, 45, 51] has focused on the fact that on NAND Flash chips (i) bits can only be cleared by incurring the overhead of erasing a large block of memory and/or (ii) individual cells can quickly wear out due to too many writes to the cell. Emerging memories, in contrast, can write arbitrary bytes in-place and system software can use the virtual-to-physical mapping to balance application writes across individual physical cells. Other prior work has targeted database query processing [18, 50, 51] or systems considerations [19, 32, 39, 53, 55, 56]. Our recent paper [12] defined an Asymmetric PRAM model that differs from the classic PRAM in charging $\omega > 1$ for writes (reads are unit cost), as well as a variety of external-memory-style models that transfer data in blocks. Write-efficient algorithms were presented for sorting, FFT and matrix multiplication. Our follow-on paper [11] presented write-efficient *sequential* algorithms for a number of fundamental problems, and defined the sequential (M, ω)-Asymmetric RAM model that combines a small symmetric-cost memory of size M with a large asymmetric-cost memory. Finally, Carson et al. [15] recently presented a number of interesting results for models with asymmetric read-write costs. Specifically, they considered (i) sequential algorithms on a model with a small symmetric memory and a large asymmetric memory, both cache-oblivious and not, and (ii) parallel algorithms on a distributed memory model where the last level of the memory hierachy on each node has asymmetric read-write costs. On the latter model, they presented upper and lower bounds for various linear algebra problems and direct N-body methods, restricted to the class of "communication-avoiding" algorithms, i.e., parallel algorithms that minimize the (unweighted) sum of reads and writes.

In this paper, we focus on parallel algorithm design under asymmetric read-write costs, extending prior work in two important ways. First, we define the Asymmetric Nested-Parallel model, which combines features of the sequential (M, ω)-Asymmetric RAM model and the popular nested-parallel model but with a distinctive memory allocation scheme. Specifically, it is comprised of small stack-allocated memories with symmetric read-write costs and an unbounded heap-allocated shared memory with asymmetric read-write costs. Stack-allocated memory is allocated by a task, available to the task and any children it forks, but is invalid when the task finishes. We show that the model, with its costs analyzed based on the computation DAG (with no notions of processors or scheduling) maps efficently onto a more concrete machine model, when using a work-stealing scheduler. In particular, the model's careful accounting for task memory usage yields good bounds on the number of writes incurred during a steal, because it can accurately capture the true working set sizes that need to be transferred. Note that our use of small amounts of symmetric memory along with the large asymmetric memory matches the expected reality of real machines.

Second, we use the new model to design the first reduced-write, work-efficient, low-span (a.k.a., low-depth) parallel algorithms for

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935767

Table 1: Results for the Asymmetric Nested-Parallel Model

problem	work	span
reduce	$\Theta(n+\omega)$	$\Theta(\log n+\omega)$
ordered filter	$\Theta(n+\omega k)$†	$O(\omega \log n)$†
list contraction	$\Theta(n)$	$O(\omega \log n)$†
tree contraction	$\Theta(n)$	$O(\omega \log n)$†
minimum spanning tree	$O(\alpha(n)m + \omega n \log(\min(\frac{m}{n}, \omega)))$†	$O(\omega \operatorname{polylog}(m))$†
2D convex hull	$O(n(\log k + \omega \log\log k))$‡	$O((\omega+\log k)\cdot \log^2 n \log\log h)$†
BFS tree	$\Theta(m+\omega n)$‡	$O(\omega \mathcal{D} \log n)$†

ω=write cost; k=output size; †=with high probability; α=inverse Ackerman; m=number of edges; ‡=expected; $\mathcal{D}$=graph diameter

reduce, list contraction, tree contraction, breadth-first search (BFS), ordered filter, and planar convex hull. For the latter two problems, our algorithms are output-sensitive in that the work and number of writes decrease with the output size. We also present a reduced-write, low-span minimum spanning tree (MST) algorithm that is nearly work-efficient (off by the inverse Ackermann function). See Table 1 for a summary of our results. All of these algorithms significantly reduce the number of writes over the best prior algorithms, e.g., by a factor of ω for list/tree contraction. While some of these results are relatively straightforward, our algorithms for tree contraction, MST, and convex hull are more novel. Our algorithms reveal several interesting techniques for significantly reducing shared memory writes in parallel algorithms without asymptotically increasing the number of shared memory reads, such as the random exponential growing-with-filtering technique used in MST and BFS, the balanced tree partitioning technique used in tree contraction, and the pre-bucketed divide-and-conquer technique used in convex hull.

2. ASYMMETRIC NESTED-PARALLEL

In this paper we use a parallel variant of the (M, ω)-Asymmetric RAM (ARAM) model [11] to analyze algorithms. The (M, ω)-ARAM is a sequential RAM with two memories: a small symmetric memory of size M for which both reads and writes take unit time, and a large asymmetric memory of unbounded size for which reads take unit time but writes take time $\omega > 1$. We extend the model to allow for multiple parallel tasks. Our goal is to allow for dynamic parallelism and to analyze algorithms using work and span (also called depth or critical path length). We therefore use the nested-parallel model [10] extended with asymmetric memory, which we refer to as the *Asymmetric Nested-Parallel* (NP) model. We include some detail on the model because there are some subtleties on how the small symmetric memory is defined in a dynamically parallel model, and some care was given to the particular formulation we give here. We also describe a more concrete machine model and map costs from the Asymmetric NP model to it.

2.1 Asymmetric NP model

In the nested-parallel model a computation starts and ends with a single ***root*** task. Each task has a constant number of registers, and runs a standard instruction set from a random access machine (RAM), except it has one additional instruction called FORK. The FORK instruction takes an integer n and creates n ***child*** tasks, which can run in parallel. Child tasks get a copy of the parent's register values, with one special register getting an integer from 1 to n indicating which child it is. The parent task suspends until all its children finish[1] at which point it continues with the registers in the same state as when it suspended, except the program counter advanced by one. We say that a computation has ***binary branching*** if $n = 2$. In the model a computation can be viewed as a (series-parallel) DAG in the standard way. We assume every instruction has a weight (cost). The ***work*** (W) is the sum of the weights of the instructions, and the ***span*** (D) is the weight of the heaviest path. The ***nesting depth*** (δ) is the maximum depth of forked tasks during the computation.

In the ***Asymmetric Nested-Parallel*** (NP) model we assume a stack allocated symmetric small memory, and a heap allocated asymmetric large memory. ***Stack allocated*** memory is memory allocated by a task, available to the task and its children, but invalid when the task finishes. It is under this model, for example, that the memory bounds for work stealing are shown [13]. ***Heap allocated*** memory is allocated by a task and can be accessed by any other task, including ancestor tasks (it is completely shared memory). Each instruction has weight one, except writes to the heap memory, which have weight $\omega \geq 1$ (in practice, $\omega \gg 1$).[2] When doing analysis of an algorithm, the term "writes" will be used to refer to the number of writes to heap allocated memory. In this paper we assume the amount of stack memory allocated by all but the leaf tasks (tasks with no forks) is constant. The amount of symmetric stack memory a leaf task can allocate is bounded by a parameter M_l. This separation into stack and heap allocated memory, and the distinction between leaf and non-leaf tasks for stack memory size, is made both because it is a convenient model for using the different memories, and also because it enables an efficient mapping onto a fixed number of processors, as justified below.

Note that the Asymmetric NP is an algorithmic cost model, as opposed to a machine model, enabling reasoning about nested-parallel computations without worrying about mapping the computation to machines. We address this scheduling issue next.

2.2 Scheduling Asymmetric NP Computations

This section shows that the algorithmic cost metrics of Asymmetric NP are sufficiently descriptive to capture the performance of Asymmetric NP computations when using good schedulers.

The Asymmetric NP model has been designed in a manner that yields an efficient mapping to an (M, ω)***-Asymmetric PRAM*** [12] machine model. In this model, there are P processors, each running its own instructions using a small symmetric ***local*** memory of size M. The processors share an unbounded asymmetric ***global*** memory, to which concurrent reads and writes are allowed. We also allow any processor to read the local memory of another processor (concurrently), but not to write to it. A request to read the local memory of another processor is viewed as requiring a write out to the global memory in order to enable the read, and hence is charged ω. On each processor any write to the global memory also takes ω time. All other instructions take unit time. For synchronizing we assume an atomic fetch-and-add to the global memory that can be performed in constant span and work linear in the number of processors.

The challenge with Asymmetric NP computations is that stack variables must be written out to global memory before tasks can be migrated to a different processor from the one that forked it.

[1] We assume, as in the RAM, there is a FINISH instruction.

[2] We treat the write cost, ω, as a parameter in our model in order to highlight its impact on algorithm design and analysis. To further highlight, we also state bounds on the number of writes in each of our algorithms. Because of our interest in practical algorithms, we seek algorithms where the constant factors hidden by the Big-O notation are small, so that the gains from using fewer writes matter.

The naïve approach would forego the stack and instead write all $O(1)$ stack variables for non-leaf tasks directly to global memory, making each FORK cost $\Theta(\omega)$. For fine-grained parallelism especially, where the number of FORKs for an algorithm with work W is $\Omega(W)$, this approach would yield running time no better than $\Omega(\omega W/P)$. In other words, one may as well consider every instruction a write if adopting the naïve scheduler.

Here we show that a variant of a work-stealing scheduler [13] achieves $O(W/P+\omega D)$ expected time when limited to binary forking. In standard (symmetric-memory) work stealing, each worker (or processor) maintains a double-ended queue called a deque of tasks that are ready to execute. Whenever a worker executes a (binary) FORK instruction, the processor continues working on the "left" child task and places the "right" child on the bottom of its deque. When a worker completes a task by executing its FINISH instruction, there are two options. If that task enabled another one, i.e., the completed task was the last outstanding child of its parent, then the worker continues working on the now-enabled parent. Otherwise, the worker removes the bottom task from its deque and executes it. If the deque is empty, the worker instead ***steals***, meaning that it chooses a random victim processor and takes the task from the top of that processor's deque if the deque is non-empty. In the event that the steal is unsuccessful, the worker will continue to attempt to steal until it successfully steals a task or the computation is finished.

In the Asymmetric NP model, working locally on a deque is cheap, and in general stack frames need not be written out. For example, if the entire computation runs sequentially on one processor, then no stack-related writes occur (assuming that the local memory is large enough to hold the stack depth).

There are still some challenges, however, most notably on steals. Because a task has access to any stack variables of its ancestor tasks, any unwritten stacks of ancestor tasks must be written out to global memory when a steal occurs. This situation is particularly challenging to analyze as it may cause steals to take time proportional to ω times the nesting depth. To cope with this challenge, Lemma 2.1 shows that a simple modification to work stealing results in at most a constant number of frames needing to be written. We assume that a steal request somehow interrupts its target task (e.g., all tasks can regularly poll to check if there are any outstanding steal requests), and if work is available, the registers for the stolen task and relevant ancestors are written to the asymmetric global memory. There is a similar potential issue when a task completes: its return value must be written to its parent task, which may no longer be local, and hence a write to global memory could be required.

LEMMA 2.1. *There exists a variant of work stealing such that on each steal (or steal attempt), only $O(1)$ stack frames are written to global memory.*

PROOF. The lemma can be achieved either by modifying work stealing or by an equivalent program transformation. The program transformation is as follows. Transform every FORK into two FORKs as follows. First FORK two tasks: the left child performs the intended FORK, which we call a forking task, and the right child is a dummy task that does nothing. The dummy task is inserted onto the deque, whereas the worker continues on to execute the forking task. (In Cilk-like work-stealing, e.g., [13], expressing the FORK as two "spawns" followed by a "sync" would automatically create a similar dummy task as a continuation from the second spawn.)

We claim that for the topmost task on the deque (i.e., the one that can be stolen), at most its parent and grandparent have not already been written out to global memory. In order for a task to be on the deque and hence stealable, it must be the right child of its parent. A simple induction shows that for any task on a deque, all right children of ancestors have either been stolen already or are on the same deque. (A similar claim is proven as Lemma 3 in [7].) Thus, a steal need only write-out the frames corresponding to the longest right-only path in the computation graph. The longest is three nodes, which we can show by cases. A forking task is always the left child and hence not stealable. A real task (i.e., one existing before the transformation) is always the child of a forking task, so it can have at most one unwritten ancestor frame: the parent forking task. A dummy task is always the right child of a real task, so stealing a dummy task could entail writing out three frames. □

THEOREM 2.2. *Consider a computation in the Asymmetric NP model with binary branching factor, W work, D span, δ nesting depth, and M_l leaf stack memory. There exists a work-stealing scheduler that executes the computation in $O(W/P+\omega D)$ expected time on a P-processor $(O(\delta)+M_l,\omega)$-Asymmetric PRAM.*

PROOF. We adopt the potential-based analysis from [7]. But we need to modify the argument to cope with the fact that (1) steals involve writing out to global memory, and (2) finishing tasks may also entail writing out to global memory, i.e., if the parent frame is in global memory.

The main idea of the original potential-based analysis [7] is as follows. On each timestep, each processor places a token in either a work bucket, if it is making progress on a task, or a steal bucket, if it is attempting to steal. Since at most W units of progress can be made on tasks, there are at most W work tokens. Moreover, every processor places a token on each timestep, so the parallel running time is the total number of tokens divided by P. It follows that if S steal attempts occur during the course of the execution, then the running time is $O(W/P + S/P)$. One can bound $S = O(PD)$ in expectation through a clever potential analysis [7]. The details are not important here, but the main idea is that each $\Theta(P)$ steal attempts are "likely" to steal the shallowest ready tasks, and hence progress is made on the span of the computation. Thus, the expected running time is $O(W/P + D)$ in the symmetric memory setting where writes are constant time.

In our case, a steal attempt does not correspond to a constant-time event. Instead, we may need to write out several stack frames. By Lemma 2.1, a steal occupies two processors for at most $O(\omega)$ timesteps. Thus, the cost of a steal token is increased by this much. We must also account for the fact that work in the computation may increase elsewhere if writes to global memory occur. In particular, assuming local memory is large enough to hold the entire stack (i.e., $O(\delta) + M_l$)), the only additional writes that occur are when a child returns to a parent task that resides in global memory, i.e., it has been stolen. This event corresponds to a more expensive write token, but the number of these heavy write tokens is bounded by the number of steals. We thus have a total running time that is $O(W/P + \omega S/P)$, where S is the number of steal attempts. The potential analysis on the number of steal attempts [7] can be applied to the augmented computation as usual, and we are left with an expected running time of $O(W/P + \omega D')$, where $D' \leq 2D$ is the span of the augmented computation created using the process described in Lemma 2.1. □

The above theorem provides justification for charging only unit cost for FORK, and for example, means that the standard reduce via a binary tree incurs only $\Theta(n + \omega)$ work instead of $\Theta(\omega n)$ work on the Asymmetric NP, as discussed in Section 3. Note also that our separate accounting for leaf stack memory in the Asymmetric NP model, and the observation that the non-leaf tasks of all the algorithms in this paper each allocate only $O(1)$ stack memory, means that the bound in the lemma is only a constant number of writes per steal, whereas without the separate accounting, it would be $O(M_l)$ writes per steal.

Bulk-Synchronous Computations. Many of the algorithms in this paper are ***bulk-synchronous*** algorithms for which there is only one level of nesting ($\delta = 1$). The root task proceeds in a sequence of R rounds. In each round i it forks n_i child tasks (each a leaf) and waits for them to finish. The root task can run arbitrary computation between such rounds. We define the iteration count I as $\sum_{i=1}^{R} n_i$. The following lemma for scheduling bulk-synchronous algorithms provides additional support for the model.

LEMMA 2.3. *A bulk-synchronous computation with arbitrary branching on the Asymmetric NP model with W work, D span, R rounds, I iteration count, and M_l leaf stack memory, can be simulated on an $(O(M_l), \omega)$-Asymmetric PRAM with P processors in $O((W + \omega I)/P + D + \omega R)$ time.*

PROOF. The idea is that the root task runs on some processor and when it gets to a fork, it sets up the registers for the children and sets a count to n. The processors then grab tasks by decrementing this count (using the fetch-and-add). When the count reaches zero idle processors quit for that round and wait for a later round when it is again set to some non-zero number. A separate counter can be used to detect when all processors are done, and the processor that detects termination can continue with the root task. The additional work done for accessing the counter is $O(\omega I)$ and the charge to the span is $O(\omega R)$ so using Brent's scheduling principle [14], $(T \leq W/P + D)$, we have the given time bounds.

Each child task takes $O(M_l)$ local memory. Since the nesting depth is one, the total memory needed by a child task is $O(M_l + 1)$. This gives the memory bound. □

3. BASIC PARALLEL PRIMITIVES

In this section we consider two basic parallel primitives. We note that some primitives inherently require as many writes as reads. For example, all prefix sums needs to write out all the sums. However, when they are used as a step for some other purpose, then the final generation of the values can be folded into whatever needs the values. This idea is used in the output sensitive filter described below.

3.1 Reduce

Summing a sequence of values with respect to an associative function $f(x, y)$ is surely the most common parallel function. Of course it only requires a single result so it should be possible to make it write efficient. In the Asymmetric NP there are two methods to do this. The first is simply to use a divide-and-conquer algorithm that recurses on the two halves in parallel, and when they return add the two results. The base case can either be a single element or $O(\log n)$ elements, and then sum those elements sequentially. A fork can be used to generate two child tasks for the calls. The interesting feature of our model is that this will only require a single write, which is to write the final answer. All other computation can be done in the constant space per task stack space. This may seem impossible since the processors need to communicate. Recall, however, that when we simulate the Asymmetric NP (with binary branching) on a machine, we account for the steals in the cost. These steals are communicating the values among processors. The divide-and-conquer algorithm leads to the following result.

LEMMA 3.1. *The reduction of n elements can be done in $\Theta(n + \omega)$ work and $\Theta(\log n + \omega)$ span using $\Theta(1)$ writes on the Asymmetric NP model.*

A second way to do a reduce is with bulk-synchronous steps. The first step forks n/ω tasks, each of which sums ω values and writes its sum to large memory. This can be repeated $\log_\omega n$ times to get the sum. The resulting algorithm does $\Theta(n + \omega)$ work and has $\Theta(\omega \log_\omega n)$ span.

3.2 Output-Sensitive Ordered Filter

Given an array A of size n and a predicate ρ on the elements of A, we want to filter out the entries $x \in A$ for which $\rho(x) = 0$, and get a new array A' of size $k \leq n$, where k is the number of elements $x \in A$ for which $\rho(x) = 1$ and A' preserves the relative order of the elements in A. In the classic nested-parallel model, this is easily done in linear work and logarithmic span by first creating a new array that holds the result of applying ρ to each element, computing a prefix sum on that array to determine the position in A' of each element to be moved, and then moving those elements. However, this requires $\Theta(n)$ writes. In the Asymmetric NP model, this number of writes can be problematic; there exist algorithms where the number of writes depends on k rather than n except for the filtering step.

Our goal is an (output-sensitive) ordered filter algorithm whose writes are proportional to the output size rather than the input size, thereby matching the lower bound on writes. We show the following result:

LEMMA 3.2. *Ordered filter on an array of size n such that k elements satisfy the given predicate can be done in $\Theta(n+\omega k)$ work and $O(\omega \log n)$ span using $\Theta(k)$ writes w.h.p. on the Asymmetric NP model.*

Our algorithm proceeds as follows: we first apply a REDUCE operation with ρ to find k, and allocate an array B of size $O(k)$. Recall that in our model, a reduce operation takes only $O(1)$ writes.

If $k \leq \frac{n}{\log n}$, then we hash the k entries, along with their indices in A into B. This takes only $O(k)$ writes and can be done efficiently in parallel, since we expect few collisions. We then sort the array by the original indices to get an array A' of size k that preserves the elements' relative order in the input array. Because there are less than $\frac{n}{\log n}$ entries, we can sort them in $O(n + \omega k)$ work and $O(\omega \log n)$ span w.h.p., using a write-efficient parallel sorting algorithm [12].

If $k > \frac{n}{\log n}$, we divide the array into k equal parts, and then in parallel process each part sequentially. This sequential filtering need only write each non-filtered element once, and it takes $O(\log n + \omega)$ span since each part is small. Finally, we concatenate the results to get our output array. This concatenation can be done in $O(\omega \log n)$ span by counting the elements in each part and using a prefix sum to find starting indices for each part's output in the final array. Since the prefix sum is only applied to the set of k parts, it uses only $O(k)$ writes.

4. LIST AND TREE CONTRACTION

In this section we introduce efficient parallel algorithms for list and tree contraction that reduce the number of writes without increasing the reads. In both problems, our goal is to divide up the problem into sub-problems that can be processed in parallel. We say that an ***equal partition*** of a structure of size n is a partition into $O(s)$ contiguous parts, each of which is of size $O(n/s)$ for some $s \leq n$. For example, the partition based on m-critical nodes in [26] is an equal partition.

The list contraction algorithm we present is relatively simple, and uses a common approach to partition the problem size using random samples. Tree contraction however, becomes more challenging when we limit the writes to main memory. We are not aware of any existing tree contraction algorithms (even sequential ones) that solve the problem using bounded local memory. Furthermore, designing a parallel version is hard because we cannot explicitly

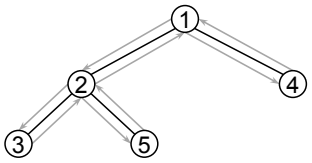

Figure 1: The Euler tour of this rooted binary tree is $(1, 2, 3, 2, 5, 2, 1, 4, 1)$, obtained by following the arrows on the edges starting at the root node 1.

record information on the tree nodes (this would require too many writes).

Previous parallel tree contraction algorithms use either a top-down approach [34, 42, 43, 48] or a bottom-up approach [26]. All of them require a linear number of writes, and we are unaware of any non-trivial modifications to these algorithms that can reduce the number of writes to $o(n)$. To solve this problem, our algorithm a uses a bottom-up approach. It is based on a new tree-partition algorithm that, instead of partitioning the tree based on sub-tree sizes as done in [26], uses the Euler tour as a tool to refine our partition after randomly selecting sample nodes. Then we prove that the partition we get is an equal partition using a surprisingly simple argument. With this partitioning of the tree, we can combine a sequential space-bounded tree contraction algorithm that we describe with any of the existing tree contraction algorithms to obtain a write-efficient parallel tree contraction algorithm in the Asymmetric NP model.

4.1 List Contraction

A linked list is a list of nodes in which each node has a pointer to the next node in the list. A segment on the linked list is defined as the elements between two given nodes. The list contraction problem is to contract a linked list of length n into a single node (possibly combining values). It has many applications, including list ranking and Euler tours [34]. Sequentially, we can just loop over all nodes by following the pointers which takes a linear number of reads and work, and a constant number of writes to main memory. In the symmetric setting ($\omega = 1$), the standard parallel approach using random mate [5] requires $O(\log n)$ span and linear reads and writes.

Our algorithm partitions the list in two steps. In the first step, each element in the list is randomly marked with probability s/n for some parameter s. Then in the second step, we start with each marked node, and in parallel, follow the list with the pointers and mark every $\lfloor n/s \rfloor$'th element. The longest chain we have to follow here has length $O((n \log n)/s)$ w.h.p., which can be shown using a Chernoff bound. Now clearly all segments between two consecutively marked nodes have size no more than $\lfloor n/s \rfloor$. With the marked nodes, we contract all segments in parallel, with each one done sequentially (terminating when the next marked node is encountered). After that, a standard symmetric version of parallel list contraction is applied on the marked nodes. Each step of the algorithm performs $O(s)$ writes.

The new list contraction algorithm takes linear work, $O(\omega + (n \log n)/s)$ span, and $O(s)$ writes in the Asymmetric NP model. This algorithm is efficient when running on a share-memory machine with $p = O(n/(\omega \log n))$ cores. In this case we can just plug in $s = n/\omega$ and get the bounds shown in Table 1. The list partition routine described above is used as a subroutine in our tree contraction algorithm described next.

4.2 Tree Contraction

The tree contraction problem is to contract a tree with n nodes into a single node (possibly combining node values), and has many applications in parallel computing [34, 42, 43]. We assume that the input is a rooted binary tree and each tree node has pointers to its parent, left child, and right child (if they exist). When the tree is viewed as a directed graph that contains two directed edges for each edge in the tree, the Euler tour [49] of the tree is an Eulerian circuit of the directed graph (see Figure 1 for an example). It can be constructed implicitly: given the current node and the previous edge, we can check whether the previous edge is from the parent or child of the current node, and according to this information, decide which edge to take next. We define a ***component*** of a tree to be a set of tree nodes that are connected. A subtree is a component, but not vice versa.

Algorithm 1 A parallel tree partitioning algorithm.

Input: A rooted binary tree T.
Output: $O(s)$ partition nodes.

1: Apply the parallel list partitioning algorithm on the Euler tour of tree T and mark no more than $O(s)$ tree nodes such that each sublist has length less than n/s
2: **for each** marked node v **do**
3: Traverse the Euler tour from the position of v's last appearance to the next marked node v'
4: Mark the highest node in this range
5: **return** all marked nodes

In the symmetric setting ($\omega = 1$) doing tree contraction sequentially in linear work is trivial, and classic parallel tree contraction algorithms [34, 42, 43, 26] take $O(\log n)$ span and $O(n)$ writes. However, many applications, such as arithmetic expression evaluation and subtree size queries on a set of tree nodes, have a sublinear output size. Thus, a natural question to ask is whether we can design a parallel tree contraction algorithm with a sublinear number of writes. We describe such an algorithm in this section.

4.2.1 A new tree partitioning algorithm

The goal of this algorithm is to find $O(s)$ partition nodes such that each tree component has size at most n/s. The high-level idea of the algorithm is to compute an equal partition of the Euler tour and mark the lowest common ancestor of each component as a partition node. The pseudocode is provided in Algorithm 1, and we explain the details below.

Since we cannot afford to store information per node, our solution is to run the parallel list partitioning algorithm described in Section 4.1 on the Euler tour of the tree. However, generating the Euler tour is too expensive, as it would require a linear number of writes. Thus, we need to simulate the parallel list partitioning algorithm. We do so as follows. First, we randomly sample $O(s)$ nodes on the tree. Then, from each sampled tree node, we follow the Euler tour in every direction in parallel. In each direction, we mark every $\lfloor n/(3s) \rfloor$'th element along the path, until we reach the next sampled node on the Euler tour. This step is similar in the list partitioning algorithm, but each node may correspond to multiple (up to three) different locations in the Euler tour, so we traverse the list from all different locations. In expectation, the distance between two samples will be at most $O((n/s) \log n)$ w.h.p. We will mark another $O(s)$ nodes during the second step. This process corresponds to Line 1 of the pseudocode.

With the $O(s)$ marked nodes, the algorithm now finds and marks $O(s)$ ***partition nodes***, which corresponds to Lines 2–4 of Algorithm 1. A partition node is the highest node (closest to the root) in a tree component consisting of tree nodes in a segment of the Euler tour. For every segment that starts with the first or second appearance of a marked interior node v, the partition node is just the first node in the segment, because the segment cannot go beyond the subtree of v and it terminates no later than the next appearance of v in the Euler

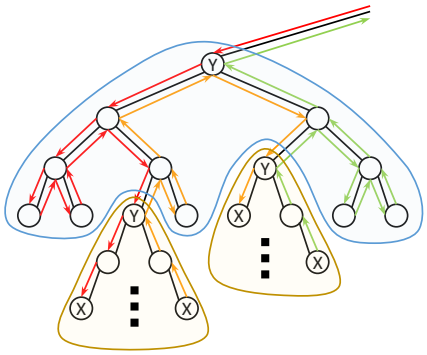

Figure 2: An example of how the tree is partitioned into components. A component consists of tree nodes from at most three segments of the Euler tour.

tour. Hence, we need to do nothing for these segments since the partition nodes are already marked. We only consider the segments that start with the last appearances of marked nodes. The partition node of each such segment can be computed with constant space by traversing through the segment: we always maintain a pointer p to the highest node so far in the traversal. We start by pointing at the first node in the segment, and whenever we go from our current top node to its parent in the traversal, we update p to point to its parent as well. When the traversal finishes, the node pointed by p is marked. It is easy to see that this marked node is the highest node in the segment.

The partition nodes provide a partitioned tree with $O(s)$ components if we form the tree components by ignoring edges from all partition nodes to their parents. We now show that this partition is an equal partition.

LEMMA 4.1. *The marked nodes generated in Algorithm 1 partition the tree such that each tree component contains at most n/s tree nodes.*

PROOF. We prove this by showing that each component consists of tree nodes from at most three segments of the Euler tour, which is shown in Figure 2. Recall that in the algorithm, we marked every $\lfloor n/3s \rfloor$'th element in the Euler tour, so showing this suffices to prove our claim.

In Figure 2, nodes marked by the list partitioning algorithm are shown with the letter "X", and the newly-added partition nodes corresponding to the highest nodes in each segment are marked with "Y". Both types of nodes are considered *marked nodes*. We claim that for each tree component rooted by either an X node or a Y node (e.g., the blue region in Figure 2), there exist at most two subtrees (e.g., the two yellow regions in Figure 2) that contain marked nodes and are rooted at nodes that are direct children of this component. More specifically, there will be at most one such subtree in each of the left and right subtrees of the root node (we will refer to these two subtrees as *left-side and right-side marked subtrees*). The segment from the last X node in the left subtree (if it exists) on the Euler tour to the first X node in the right subtree (if it exists) marks the root of this component. To see that there is at most one subtree on each side, assume for the sake of contradiction that there are two marked nodes on the same side of the root but not in the same marked subtree (yellow region). Then the lowest common ancestor of these two nodes will be marked as a Y node. This node is assumed to be in this component (the blue region) but actually it will be marked and removed from this component, which leads to a contradiction.

Hence, each component only consists of nodes from at most three segments from the Euler tour. The first segment is the one that enters this component in the Euler tour, and ends at the first X node (based on the Euler tour) in the left-side marked subtree, which is shown as the red arrows in Figure 2. The second segment is illustrated as the orange arrows, starting from the last X node in the left-side marked subtree and ending in the first X node in the right-side marked subtree. The last segment, shown as the green arrows, is symmetric to the first segment and leaves this part from the last X node. If there are no marked node in the left side, then the first and second segments are merged into one, and similarly on the right side. Thus each component consists of vertices from at most three segments of the Euler tour.

Since we can guarantee that each segment has size no more than $n/3s$, each tree component contains no more than n/s nodes. □

The overall cost is the sum of the cost of partitioning the Euler tour and the cost of traversing a subset of the segments (starting for the up-edge of each X node). This takes linear work, $O(s)$ writes and $O((n/s)\log n)$ span w.h.p.

Remark: Notice that our tree partition is done by removing (actually ignoring) edges but not vertices (as done in [26]) since this is more efficient on both work and writes in practice. This algorithm also works on all constant-degree trees. However, for an arbitrary tree our algorithm would not work. In fact, if the given tree is a star, then there does not exist any equal partition of non-constant ($\omega(1)$) size, which means that no top-down approaches by tree partitioning will give a write-efficient solution in this case. Therefore, a preprocessing step that converts the input to a binary tree with linear writes is required before running our algorithm, but if the tree contraction is run multiple times on a given tree, there is still an advantage in using this algorithm.

4.2.2 A sequential algorithm

We now discuss a sequential and space-bounded tree contraction algorithm that will be used as a subroutine in our parallel algorithm. Previous tree contraction algorithms [42, 43, 48, 26] take either linear space or linear writes to the main memory, which is too costly in the asymmetric setting. Instead we would like to design an algorithm that uses a small amount of local (small) memory and performs no writes to main memory. The tool we use is the tree partitioning algorithm discussed in the previous section, which can be used to partition a tree into components of size no more than n/s using $O(s)$ writes. Our algorithm then contracts each tree component down to a single node, and after that we apply a standard tree contraction algorithm on the marked nodes. To restrict the number of components as well as the size of the components so that all intermediate results fit into a small memory and require no writes, the we recursively apply the tree partitioning algorithm until each component fits in small memory. Suppose that the cutoff size for the base case of the recursion is c. Then the required small memory size is $O(s\log_s(n/c)+c)$, and the work and number of reads is $O(n\log_s(n/c))$. By setting s to either $O(\epsilon n^\epsilon)$ or some constant greater than 1, we obtain the following lemma.

LEMMA 4.2. *The sequential algorithm presented above contracts a tree of size n requiring $O(1)$ writes, and using a small memory of size $O(n^\epsilon)$ and linear work, where $0 < \epsilon < 1$, or a small memory of size $O(\log n)$ and $O(n\log n)$ work and reads.*

4.2.3 A parallel algorithm

We now describe a parallel tree-contraction algorithm that uses the tree partitioning algorithm and sequential tree contraction algorithm as subroutines.

The high-level idea for the parallel algorithm is to first partition the tree into small, almost equal-sized components using the tree

partitioning algorithm, and then contract each component independently in parallel using the sequential contraction algorithm. Finally, we use a standard parallel tree contraction algorithm to contract the remaining nodes. This step requires a number of writes proportional to the number of remaining nodes, which is much smaller than the original tree size. The algorithm takes s as a parameter and consists of three steps as described below.

Step 1: Tree Partitioning. The first part of the algorithm computes a tree partition such that each component has size $O(n/s)$, where s is a parameter of the algorithm. This step requires $O(n)$ work, $O(\omega + (n \log n)/s)$ span w.h.p. and $O(s)$ writes, if implemented by Algorithm 1.

Step 2: Tree Contraction on Components. With the partition nodes computed in Step 1, we now have a set of components each with size at most $O(n/s)$. Since the tree components are themselves trees, we can apply our sequential tree contraction algorithm on each of them. The contraction is restricted to be inside each component by not contracting any partition node. Also, we leave the root node of each component uncontracted. After contraction, each component's root has at most one left and one right child, which is either the marked child (the Y nodes in the yellow regions in Figure 2) or the a descendant of the root that is in the component (if no marked child on the side). Hence, if the local memory has size $O((n/s)^\epsilon)$, this step takes linear time and $O(n/s)$ span, and yields an intermediate contracted tree with no more than $O(s)$ nodes.

Step 3: Contraction of Remaining Nodes. In the last step, we apply any existing parallel tree contraction algorithm with linear work and logarithmic span to contract the tree generated from the last step to a single node. This step costs $O(s)$ reads, writes and work, and $O(\omega \log n)$ span.

To reduce the number of writes without increasing the asymptotic work complexity, we choose $s = n/\omega$. Hence, the size of the components is at most ω. The following theorem gives the cost of our parallel tree contraction algorithm.

THEOREM 4.3. *Algorithm 1 can be used to contract a tree with size n using $O(n)$ work, $O(\omega \log n)$ span and $O(n/\omega)$ writes with $O(\omega^\epsilon)$ local memory in the Asymmetric NP model.*

5. MINIMUM SPANNING TREES

This section extends Karger, Klein and Tarjan's (KKT) [35] sequential linear-work randomized algorithm for minimum spanning tree/forest. At a high level, their algorithm proceeds as follows: Randomly sample half of the edges, and calculate a minimum spanning forest on these sampled edges. Use the sampled forest to filter out edges that cannot be part of the overall minimum spanning forest. Specifically, identify all edges $e = (x, y)$ of the graph such that e is the heaviest edge on the cycle it closes in the sampled forest. These edges are discarded. Next, recurse on the remaining graph and perform a constant number "Borůvka steps" to reduce the number of nodes in the graph. The proof that this algorithm runs in linear work hinges on two main facts: first, that the filtering can be done in linear work, and second, that the number of edges filtered out is large in expectation. We will use the following definitions in this section.

DEFINITION 1. *For a given tree T in the graph, and two vertices $x, y \in V$, the path connecting x and y in T (if such a path exists) is denoted by $\tau(x, y)$.*

DEFINITION 2. *An edge $e = (x, y)$ is said to be **heavy** with respect to a tree T on the graph if e closes a cycle in T and it is the heaviest edge in that cycle. That is, $w(e) \geq \max(w(e') \mid e' \in \tau(x, y))$. Any edge that is not heavy with respect to T is said to be **light** with respect to T.*

Recall the well-known *cycle property* of MSTs: a tree T is an MST in a graph G if and only if every non-tree edge closes a cycle in T and is the maximum weight edge on that cycle. Therefore, if all edges of a graph not in T are heavy, then T is an MST of that graph.

The filtering-out of heavy edges is achieved via minimum spanning tree verification algorithms. These algorithms take a tree and a graph as input and determine whether the tree is an MST in the graph. They operate by labeling each edge as light or heavy with respect to the tree. There is extensive work on the verification of minimum spanning trees, and several algorithms are known to operate in linear work [21, 37, 31]. The KKT algorithm then throws out any edges labelled heavy by the verification algorithm, and recurses on the remainder of the graph.

To be efficient, the KKT algorithm requires that when a minimum spanning forest is built on a random subset of the graph, a large fraction of edges in the remaining graph can be filtered out. They show that this holds using the following lemma:[3]

LEMMA 5.1 (SAMPLING LEMMA). *For a random subset $R \subseteq E$ of size r, and a random subset $S \subseteq E$ of size s, the expected number of edges in S that are light with respect to the MST of R is less than sn/r.*

Using this lemma, Karger et al. prove that their algorithm requires $O(m)$ work with high probability.

5.1 Write-efficient MST

Our goal is to find an MST algorithm that minimizes the number of writes performed without significantly increasing the number of reads. A slight modification of Borůvka's algorithm yields an algorithm which executes in $O(m \log n + \omega n)$ work in the (M, ω)-ARAM model [11]. That is, we can achieve the optimal number of writes ($O(n)$) using $O(m \log n)$ reads. However, we want to find an algorithm that more closely matches the optimal work of the algorithm of Karger et al. [35].

We present an MST algorithm that uses the KKT algorithm as a subroutine, and requires $O(\alpha(n)m)$ reads (where $\alpha(\cdot)$ is the inverse Ackermann function) and $O(n \log(\min(m/n, \omega)))$ writes, resulting in $O(\alpha(n)m + \omega n \log(min(m/n, \omega)))$ work.

The algorithm is iterative, and proceeds as follows. It begins by taking an $O(n)$ sized random sample of the edges and running KKT on them to find a minimum spanning forest of the sampled graph. Then, in each round, it takes a random sample that is twice as large as the previous one, filters out the edges in the sample that are heavy with respect to the most recently calculated spanning forest, and then runs KKT again on the remaining part of the sample. In each such round, with high probability, we will only be left with $O(n)$ edges from the sample that pass the filtering, and so each round will need $O(n)$ writes. The algorithm proceeds in this way until the final sample includes all edges in the graph. The pseudocode for this algorithm is presented in Algorithm 2.

5.2 Analysis

We start with a sample set S of edges with size $\max(2n, m/\omega)$, and run the KKT algorithm on this sample to calculate a minimum spanning forest on it. We then double the size of the sample set in

[3]The statement of this lemma is a slight variation of the version given in [17] and is different from the original paper [35].

Algorithm 2 Write efficient MST algorithm

Input: A graph $G = (V, E)$.
Output: An MST with a set T of edges.

```
1: sampleSize ← max(2n, m/ω)
2: Edge set T ← {}
3: while sampleSize < m do
4:     The set of light edges S_L ← {}
5:     for s ← 1 to sampleSize do
6:         Randomly pick an edge e ∈ E
7:         if e cannot be filtered with respect to T then
8:             S_L ← S_L + {e}
9:     T ← KKT(S_L ∪ T)
10:    sampleSize ← 2 · sampleSize
11: S_L ← {}
12: for all e ∈ E do                    ▷ Take all edges in final round
13:     if e cannot be filtered with respect to T then
14:         S_L ← S_L + {e}
15: T ← KKT(S_L ∪ T)
16: return T
```

each round and recalculate, until all edges are processed. However, to save writes, instead of storing all the samples, we use an online filtering algorithm to throw out the edges that are heavy, and only keep the light sample edges. We sample slightly (a constant factor) more than our desired sample size to account for collisions. Note that once the sample size is linear in m, we can sample by flipping a coin with appropriate probabilities for each edge, without increasing the algorithm's total work. We define the edge set S_L to be the subset of S that contains light edges (with respect to the current spanning forest in this round), and the edge set S_H to be the subset consisting of heavy edges. Clearly, $S = S_H \cup S_L$.

LEMMA 5.2. *At the end of round i of the algorithm, we have a minimum spanning forest on $\Theta(2^{i-1}\max(n, m/\omega))$ edges of the graph in expectation.*

PROOF. In round i, we have at least $\Theta(2^{i-1}\max(2n, m/\omega))$ sample edges in expectation, but only build a minimum spanning forest using the edges in S_L, along with the tree we already have. Let T be the set of edges in the current forest, as shown in the pseudocode. Note that by the definition of heavy edges, T is a minimum spanning forest in the graph whose edge set is $S_H \cup T$. Therefore, the MST of $T \cup S_L$ is also a minimum spanning forest on $T \cup S_L \cup S_H$. □

Therefore, at the end of round $\lceil \log_2(\min(m/n, \omega)) \rceil$, we have an MST on the entire graph, and we are done. Note that in the last round, we simply consider all of the edges (without sampling) to ensure that we've seen every edge.

LEMMA 5.3. *In every round, the expected size of S_L, the number of edges that pass the filtering, is $\Theta(n)$.*

PROOF. By Lemma 5.2, the MST used to filter edges in round i is a minimum spanning forest on at least $c_1 2^{i-1} n$ edges in expectation for some constant c_1. The sample size in round i is $c_2 2^i n$ in expectation for some constant c_2. By the Sampling Lemma, the expected size of S_L in round i is less than $\frac{sn}{r} = \frac{c_2(2^i n)n}{c_1 2^{i-1} n} = \Theta(n)$. □

By Lemmas 5.2 and 5.3, it is easy to see that, excluding any work needed for the filtering, the total number of writes required for this algorithm is $O(n \log(\min(m/n, \omega)))$, and the number of reads is $\sum_{i=1}^{\lceil \log_2(\min(m/n,\omega)) \rceil} \Theta(2^i n) = O(m)$.

Ideally, we would like the filtering step to take no more than $O(n)$ writes per round, and a constant number of reads per edge. There are several MST verification algorithms that take work linear in the number of edges [21, 37, 31]. However, all of these algorithms also take $O(m)$ writes, and are therefore not suitable for us. Alon and Schieber [4] present an online algorithm for minimum spanning tree verification that operates in $O(n)$ preprocessing work, and then $\alpha(n)$ work per queried edge. The queries are done through a look-up in the data structure built in the preprocessing stage, and require no writes. Using their algorithm allows us to execute the entire write-efficient MST algorithm in $O(n \log(\min(m/n, \omega)))$ writes and $O(\alpha(n)m)$ reads.

Alon and Schieber also prove a matching lower bound for the problem of answering online tree product queries, which is a generalization of the MST verification problem. However, they show this lower bound looking at the worst case query. It may be possible to improve upon this result by considering the query time amortized over all of the edges.

5.3 Parallel Analysis

We can parallelize each of the steps of the algorithm. Clearly, the sampling of edges can be done in parallel in constant span. We then use a parallel version of Alon and Schieber's algorithm [4] to filter out heavy edges. This takes polylogarithmic span. Cole et al. [20] presented a parallel version of the KKT algorithm that takes linear work and polylogarithmic span, which we can use instead of the sequential version whenever we call KKT. So simply by using the parallel versions of these algorithms, we achieve a work-efficient polylogarithmic span algorithm (we only sample $O(\log(\min(m/n, \omega)))$ times, and the span is only increased by a factor of $O(\log(\min(m/n, \omega)))$).

However, we need to be precise with the number of writes required in the parallel version. After using Alon and Schieber's algorithm to check whether each sampled edge can be filtered out, we need to pack out the edges that passed the filtering to use them in the next round of the algorithm. A standard packing algorithm would execute a number of writes proportional to the total size of the sample, which is too many writes for us. For this, we use the output sensitive filter algorithm presented in Section 3.2.

We thus obtain the following theorem:

THEOREM 5.4. *Given a graph G with m edges and n vertices, an MST of G can be found in $O(\alpha(n)m + \omega n \log(\min(m/n, \omega)))$ work, $O(\omega \text{ polylog}(n))$ span and $O(n \log(\min(m/n, \omega)))$ writes w.h.p. in the Asymmetric NP model, where $\alpha(\cdot)$ is the inverse Ackerman function.*

6. OUTPUT-SENSITIVE CONVEX HULL

The ***planar convex hull*** problem takes as input a set of points in 2D and generates the smallest polygon (represented as a list of segments) that contains all of the points. The fastest algorithms for computing a convex hull either require $O(n \log n)$ work if the algorithm is insensitive to the output size, or $O(n \log h)$ work, where h is the number of points on the hull. Naive implementations of these algorithms would require $O(n \log n)$ or $O(n \log h)$ reads/writes, respectively, while the minimum number of writes required is much lower since only the h points and segments on the hull need to be written. Our goal in this section is to develop work-efficient parallel algorithms where the number writes is asymptotically lower than the number of reads.

Several algorithms first sort the points by increasing x-coordinate, and then apply a $O(n)$ work step on the sorted points to compute the hull (see, e.g., [30, 6]). We note that we can trivially obtain an

Algorithm 3 OUTPUT-SENSITIVE (UPPER) CONVEX HULL

Input: A set of n two-dimensional points in general position.

1: Place the points into h buckets each of size $O(n/h)$, where the x-coordinates of all points in bucket $i \in [0, \min(h-1)]$ are less than the x-coordinates of all points in buckets $j > i$.
   (a) Pick $\Theta(h \log n)$ random samples, sort them, and use every $(\log n)$'th sample as a splitter.
   (b) Have the remaining points each do a binary search on the splitters to determine which bucket they belong to.
   (c) Use prefix sums to determine and appropriate offsets into buckets for each point.
   (d) Have all points write to the appropriate offset into their bucket.
2: If there is only one bucket B, search up the tree of bridges and perform one of the following:
   (a) If all points in the buckets are on or below a bridge, then do nothing.
   (b) If there are points in B not covered by a bridge, then apply a standard output-sensitive convex hull algorithm on an input containing points in the bucket and the points forming the bridges $Br_{B,L}$ and $Br_{B,R}$.
3: Split points into two sets, L and R, where L contains points in the first $\lceil h/2 \rceil$ buckets, and R contains the remaining points.
4: Find the bridge between L and R.
5: Recursively apply Steps 2–5 on each of L and R, storing the bridge computed in each sub-problem as the left and right child, respectively, of the bridge in Step 4.
6: Obtain final solution by traversing down the tree of bridges.

algorithm with $O(n \log n)$ reads and $O(n)$ writes, by first using a write-efficient sort [12], followed by the same post-processing step that takes $O(n)$ reads/writes. The sort can be done in $O(\omega \log n)$ span w.h.p. [12], and the post-processing step can be done in parallel in $O(n)$ work and $O(\omega \log n)$ span [29].

6.1 An output-sensitive algorithm

Obtaining a write-efficient output-sensitive convex hull algorithm with $O(n \log h)$ work requires more effort because we can no longer directly apply a comparison sort. We now describe how to obtain an algorithm with $O(n \log h)$ reads and $O(n \log \log h)$ writes.

Our algorithm uses divide-and-conquer and borrows ideas from [38, 16]. We first assume that we know the value of h and that $h = O(n/\log n)$ (to make oversampling work); we will remove these assumptions later. We also assume without loss of generality that no points have the same x-coordinate. We describe how to compute the upper hull (the hull above the line from the leftmost point to the rightmost point) and the lower hull can be computed analogously. The main steps of the algorithm are shown in Algorithm 3.

The algorithm is a divide-and-conquer algorithm but to avoid data movement we approximately pre-sort the points. In particular, we split the points into h buckets each of size $O(n/h)$, where the x-coordinates of all points in bucket $i \in [0, h-1]$ are less than the x-coordinates of all points in buckets $j > i$. By picking $\Theta(h \log n)$ samples, sorting them, and using every $(\log n)$'th element as a splitter, the buckets can be shown to have size $O(n/h)$ w.h.p. using Chernoff bounds. This step is described as Step 1 of Algorithm 3.

If the input contains a single bucket, then we have reached the base case (Step 2), which we will describe how to handle shortly. Otherwise, the algorithm splits the points into approximately two halves, L and R (Step 3). This step requires no data movement, since the points have already been placed into their respective buckets.

We then find the ***bridge*** of the upper hull between L and R, which is a line passing through a point in each set such that all points lie below the line (Step 4). The bridge can be found by solving the following two-dimensional linear program, where the bridge is the line $y = \alpha x + \beta$ and x_{mid} is the x-coordinate of a vertical line between L and R (which can be computed as a value arbitrarily close to the x-coordinate of the splitter element [41]):

$$\begin{aligned} \text{minimize} \quad & \alpha x_{mid} + \beta \\ \text{subject to:} \quad & \alpha x_{p_i} + \beta \geq y_{p_i} \quad \forall i \in L \cup R \end{aligned}$$

We describe how to solve 2D linear programs write-efficiently in Section 6.3.

The bridge found in Step 4 is stored as the root of a tree of bridges, and we recursively compute the tree of bridges on each of L and R in Step 5.

We now describe the base case, when there is only a single bucket B. The bucket searches up the tree of bridges, and considers any bridge whose x-range intersects with the bucket's x-range. A bridge can either lie on or above all points in B, have only a left endpoint in B, or have only a right endpoint in B. If we find any bridge that covers all of B, then no new convex hull edges will be generated from B and we are done. Otherwise, we find the one or two bridges that cover the most points in the bucket, and solve a subproblem with points in B and the up-to-two bridges using a standard $O(n \log h)$-work output-sensitive convex hull algorithm.

Call the set of the bridges with a left endpoint in the bucket $Br_{B,L}$, and the set of bridges with a right endpoint in the bucket $Br_{B,R}$. We wish to include the bridge in each set that covers the most points in the bucket. For a bridge $b \in Br_{B,L}$, let p_b be the endpoint of b in bucket B. The bridge in $Br_{B,L}$ that satisfies this criteria is a bridge b with the minimum value of x_{p_b}, since all points in B are below the bridge, and bridges $b \in Br_{B,L}$ cover all points in B with x-coordinate greater than x_{p_b}. If there are ties, then any bridge with x_{p_b} equal to the minimum value suffices. Similarly, the bridge in $Br_{B,R}$ that satisfies this criteria is a bridge b with the maximum value of x_{p_b}. Both of these bridges can be found during the search up the tree of bridges.

To obtain the final solution (Step 6), we start at the root of the tree of bridges, include the bridge in the solution, and recursively include into the solution the bridges in the descendants of the root that have not been already covered by a previously included bridge. To determine whether to include a bridge, we can search up the tree to see whether it has been covered. If the base case is reached, then all bridges formed from the set of points in the bucket are included.

Cost analysis.

We now analyze the cost of Algorithm 3. Step 1a can be done using write-efficient sorting [12] in $O(\omega \log n)$ span w.h.p. using $O(n \log h)$ reads and $O(n)$ writes. Step 1b takes $O(n \log h)$ reads, $O(n)$ writes, and $O(\omega + \log h)$ span. Steps 1c takes $O(n)$ reads and writes, and $O(\omega \log n)$ span. Finally, Step 1d, takes $O(n)$ reads and writes, and $O(1)$ span. So the cost for this pre-processing is $O(n \log h)$ reads, $O(n)$ writes and $O(\omega \log n)$ span w.h.p.

The number of levels of recursion of Steps 2–5 is $O(\log h)$ w.h.p. since there are h buckets at the beginning and each sub-problem contains half as many buckets.

Step 2 takes $O(\log h)$ reads, $O(1)$ writes, and $O(\omega + \log h)$ span to find $Br_{B,L}$ and $Br_{B,R}$ per bucket. Summed over all buckets, this takes $O(h \log h)$ reads, $O(h)$ writes and $O(\omega + \log h)$ span. Each sub-problem solved using a standard output-sensitive algorithm contains $O(n/h)$ points, and only generates segments on the upper convex hull of the original point set, since we included the bridges coming in from both sides of the bucket. The total number of

operations is $\sum_{i=1}^{h} O((n/h)\log(1+h_i))$, where h_i is the number of points on the hull in bucket i and $h = \sum_{i=1}^{h} h_i$. The sum is maximized when all h_i's are equal, giving a total of $O(n)$ reads and writes. The algorithm of Kirkpatrick and Seidel [38] can be parallelized to take $O(n \log h)$ work and $O(\omega \log^2 n)$ span [27]. Note that the algorithm that we apply on the buckets on must take $O(n \log h)$ work overall for the value of h that we guessed, not the number of points on the actual convex hull. Once the amount of work done on the buckets exceeds cn for some constant c, we can assume that our guess of h is wrong, and terminate. To keep track of the work, we can modify the Kirkpatrick and Seidel algorithm that we use on the buckets to increment a shared counter in global memory whenever an operation is performed, and also check the counter before performing an operation and if it is above cn then terminate. The total number of reads/writes for maintaining the counter is cn, which is within our bounds.

Step 3 requires constant work/span since the points are pre-sorted, and Step 4 requires $O(n)$ reads, $O(\log n)$ writes, and $O((\omega + \log n)\log n)$ span w.h.p. as shown in Section 6.3. There are $O(\log h)$ levels of recursion so the overall number of reads in this step is $O(n \log h)$ in expectation. The number of writes from this step satisfies the recurrence $W(n) = 2W(n/2) + O(\log n)$ which solves to $O(n)$. The total span is $O((\omega + \log n)\log n \log h)$ as each of the two recursive calls in Step 5 can be executed in parallel.

In Step 6, the searches up the tree take $O(\log h)$ reads and $O(1)$ writes for a total of $O(h \log h)$ reads and $O(h)$ writes. The span is $O((\omega + \log h)\log h)$.

Overall, the algorithm requires $O(n \log h)$ reads, $O(n)$ writes, and $O((\omega + \log h)\log^2 n)$ span w.h.p.

As done in [16], to remove the assumption that we know h, we will repeatedly guess h and apply the above algorithm until our guess is above the true value of h. On the i'th application of the algorithm, our guess will be $h^* = 3^{2^i}$, so in total we require $O(\log \log h)$ iterations until $h^* \geq h$. The number of reads per iteration is $\sum_{i=0}^{O(\log \log h)} O(n \log 3^{2^i}) = O(n \log h)$. The number of writes is $O(n)$ per iteration, for a total of $O(n \log \log h)$. Finally, the span is $O((\omega + \log h)\log^2 n \log \log h)$ w.h.p. To remove the assumption that $h = O(n/\log n)$, once our guess of h exceeds $cn/\log n$ for some constant c we call the output-insensitive algorithm described earlier, which takes $O(n \log n) = O(n \log h)$ reads, $O(n)$ writes, and $O(\omega \log n)$ span w.h.p. We obtain the following theorem.

THEOREM 6.1. *A planar convex hull can be computed with $O(n(\log h + \omega \log \log h))$ expected work, $O((\omega + \log h)\log^2 n \log \log h)$ span w.h.p., and $O(n \log \log h)$ writes w.h.p. under the Asymmetric NP model.*

6.2 Another output-sensitive algorithm

Here we describe an algorithm with $O(nh)$ reads, $O(n)$ writes, and $O((\omega + \log n)\log n)$ span w.h.p., obtained by modifying the algorithm of Kirkpatrick and Seidel [38]. Their original algorithm finds a bridge on the two halves of the points, uses the bridge to filter out a constant fraction of the points, and recursively finds the hull of the remaining points on each half. The number of levels of recursion is $O(\log h)$ and the number of sub-problems solved is h. Their algorithm as described takes $O(n \log h)$ reads and writes, and $O(\log n \log h)$ span.

Our goal is to reduce the number of writes. Instead of moving the points such that points for a sub-problem are contiguous, we just inspect all of the points each time we need to find a bridge in a sub-problem. Furthermore, we do not filter out any points. Therefore, each time we need to find the bridge, we use the 2D linear programming algorithm in Section 6.3, taking $O(n)$ reads and $O(\log n)$ writes. The number of times we solve the 2D linear program is h, giving a total of $O(nh)$ reads and $O(h \log n)$ writes. We can find the splitters that divide the points approximately evenly, which is needed for the linear program, by taking a random sample of $\min(n, h \log n)$ elements at the beginning, sorting them, and using every $(\log n)$'th element as a splitter, as done in our first algorithm. This takes $O(n \log h)$ reads, $O(n)$ writes, and $O(\omega \log n)$ span w.h.p. in total. Since the 2D linear programming algorithm requires a randomized order, we generate a random permutation of the constraints at the beginning, and use it throughout the algorithm, taking $O(n)$ reads and writes and $O(\omega \log n)$ span. The two recursive calls can happen in parallel, so the span is $O((\omega + \log n)\log n \log h)$ w.h.p. To reduce the overall number of writes to $O(n)$, if the algorithm has not terminated after $O(\log \log n)$ levels of recursion (after $O(n \log n)$ reads and $O(\log^2 n)$ writes have been done in solving the LPs), we can switch to the output-insensitive algorithm that takes $O(n \log n)$ reads and $O(n)$ writes. This gives the following theorem.

THEOREM 6.2. *A planar convex hull can be computed with $O(n(\min(h, \log n) + \omega))$ expected work, $O((\omega + \log n)\log n \log h)$ span w.h.p., and $O(n)$ writes under the Asymmetric NP model.*

6.3 2D linear programming for convex hull

We use Seidel's randomized incremental algorithm [47] for 2D linear programming, which we first review. We assume there are bounding planes such that the solution is not unbounded. The algorithm inserts the constraints incrementally, in a random order, maintaining the optimum point so far. For each added constraint it checks if it makes the current optimum infeasible; if so, it finds the best feasible solution satisfying all constraints added so far, and otherwise the old optimum point is kept. The new optimum must lie on the halfplane defining the newly added constraint, and can be found by finding the best point among all intersections between previous constraints and the newly added constraint (a one-dimensional linear program). If the points are given in random order, then the probability that the i'th constraint makes the optimum point infeasible is at most $2/i$. Checking against all previous constraints to find a new optimum takes $O(i)$ work, so the overall work is $\sum_{i=1}^{n} O(i) \cdot (2/i) = O(n)$ in expectation.

We assume that the constraints are given in a random order. For our algorithms, this requirement can be satisfied by generating a random permutation of the points at the beginning in $O(\omega \log n)$ span w.h.p. and $O(n)$ reads and writes [42], and using it to order the constraints throughout the algorithm.

The only time writing is required is when a new optimum needs to be found. In expectation, this will happen at most $\sum_{i=1}^{n} 2/i = O(\log n)$ times. A high probability bound can be obtained using Chernoff bounds. The 1D linear program, which involves maximum/minimum operations, can be solved using the reduce primitive from Section 3 using $O(1)$ writes, so the number of writes is $O(\log n)$ overall. To parallelize this, we add the constraints in rounds, where the k'th round processes the next 2^k unprocessed constraints. Processing a set of constraints in a round involves repeatedly finding the lowest-indexed constraint that makes the current optimum infeasible (a tight constraint), finding a new optimum by adding the tight constraint, and removing it and all lower-indexed constraints from the set. Finding the lowest-indexed constraint can be done with a reduce, and as discussed before finding the new optimum can also be done with a reduce, taking $O(2^k)$ reads and $O(1)$ writes. The expected number of times this process is repeated is equal to the number of constraints in the set that cause a new optimum to be computed, which on round k is at most

$\sum_{j=2^k}^{2^{k+1}-1}(2/j) \leq 2^k \cdot (2/2^k) = O(1)$, so each round takes $O(2^k)$ reads and $O(1)$ writes. The span per round is $O(\omega + \log n)$ for reduce. There are a total of $O(\log n)$ rounds until all constraints are processed. This gives a parallel algorithm for solving 2D linear programming, given constraints in a randomized order, that requires $O(n)$ reads in expectation, $O(\log n)$ writes w.h.p., and $O((\omega + \log n)\log n)$ span w.h.p. Note that for convex hull, we are reusing the permutation throughout the algorithm, so the probabilities are not independent among sub-problems. However, the span and number of writes are w.h.p., so we can take a union bound on the failure probability over the h sub-problems and still get a high probability bound. The number of reads is an expectation and so is unaffected. This gives the following lemma.

LEMMA 6.3. *When constraints are given in a random order, 2D linear programming can be done with $O(n + \omega \log n)$ expected work, $O((\omega + \log n)\log n)$ span w.h.p., and $O(\log n)$ writes w.h.p. under the Asymmetric NP model.*

7. BREADTH-FIRST SEARCH

The *breadth-first search* (BFS) problem takes as input an unweighted graph $G = (V, E)$ and a source vertex r, and returns breadth-first search tree rooted at r containing all vertices reachable from r. This section describes a parallel write-efficient BFS algorithm. We will use the notation $n = |V|$ and $m = |E|$. The standard sequential BFS algorithm is write-efficient, but not parallel. It requires $O(m + \omega n)$ work, including $O(n)$ writes (the minimum number of writes required for BFS). On the other hand, the standard parallel level-synchronous BFS algorithm of [10] is not write-efficient, requiring $O(m)$ writes. The algorithm explores the graph in parallel, where round i visits all vertices at a distance i away from r (we call the vertices newly explored in round $i-1$ the *frontier* for round i). Each frontier vertex visits and writes to all of its unexplored neighbors in parallel, which causes additional writes when multiple frontier vertices attempt to visit the same vertex simultaneously. This algorithm uses $O(\omega(m+n))$ work, $O(\omega\mathcal{D}\log^* n)$ span w.h.p. (using approximate compaction [28]), and $O(m+n)$ writes, where $\mathcal{D}$ is the diameter of the input graph.

We now present an algorithm for BFS that runs in $O(m + \omega n)$ work and $O(\omega\mathcal{D}\log n)$ span using only $O(n)$ writes in expectation. The algorithm works like level-synchronous BFS, but makes use of exponential delaying algorithm when visiting the vertices in a round to reduce collisions (and writes) on a shared neighbor. In exponential delaying exploration takes place in iterations, where on each iteration we process a fraction of the vertices on the frontier. We first randomize the order of the frontier vertices, and then on the first iteration we process the first vertex and on iteration $i > 1$, we process the next 2^{i-2} vertices on the frontier. During the exploration process, a vertex checks to see if its neighbor has been visited and only updates that neighbor if it was not visited in a prior round or a prior iteration in the same round.

We now show that the number of writes for this algorithm is $O(n)$ in expectation. Consider a vertex v that is adjacent to the current frontier and has not yet been visited. Let F be the number of vertices on the frontier (without loss of generality, assume it is a power of 2), let $N_F(v)$ be the number of in-neighbors v has on the frontier. We will use $\alpha = 1 - (N_F(v)/F)$ for notational convenience. The probability that v is first visited in iteration 1 is α and in iteration $i > 1$ is:

$$(1-\alpha^{2^{i-2}})\alpha\prod_{j=2}^{i-1}\alpha^{2^{j-2}} = (1-\alpha^{2^{i-2}})\alpha^{2^{i-2}}$$

The expected number of vertices that attempt to visit v in iteration 1 is $N_F(v)/F = 1-\alpha$ and in iteration $i > 1$ is $2^{i-2}(1-\alpha)$. Summing the expectations over all iterations gives:

$$(1-\alpha)\alpha + \sum_{i=2}^{\log F} 2^{i-2}(1-\alpha)(1-\alpha^{2^{i-2}})\alpha^{2^{i-2}}$$
$$< O(1) + \sum_{i=2}^{\log F} 2^{i-2}\alpha^{2^{i-2}} \;=\; O(1)$$

where we use $\sum_{x=1}^{\infty} xa^x = O(1)$ for $0 \leq a < 1$. This shows that the expected number of writes to a vertex is $O(1)$, and the expected number of writes overall is $O(n)$.

Randomly permuting the vertices sums to linear work overall and $O(\omega \log n)$ span w.h.p. per round [42]. The $\log n$ iterations used in the exponential delaying also contributes $O(\omega\mathcal{D}\log n)$ to the span per round. Thus the overall span is $O(\omega\mathcal{D}\log n)$. The number of reads remains $O(m+n)$, which gives the following theorem.

THEOREM 7.1. *For a graph with n vertices, m edges, and diameter $\mathcal{D}$, our write-efficient breadth-first search algorithm requires $O(m + \omega n)$ work in expectation, $O(\omega\mathcal{D}\log n)$ span w.h.p., and $O(n)$ writes in expectation on the Asymmetric NP model.*

8. CONCLUSION

In this paper we have studied parallel algorithms that are efficient in terms of how many writes they do to main memory. The research is motivated in large part by new memory technologies in which writing is more expensive than reading. We believe, however, that reducing writes is of interest even without this motivation, both out of theoretical interest, and because there are other practical reasons to reduce writes (e.g., reducing cache-coherence traffic). We defined the Asymmetric Nested-Parallel model that enables algorithm analysis in the popular work-span framework while accounting for the extra cost of writes. Our bounds for a slight variant of work stealing schedulers showed that the Asymmetric NP bounds map directly and efficiently to the (M, ω)-Asymmetric PRAM machine model.

Our paper presented several novel techniques for designing write-efficient parallel algorithms. One is to process a geometrically increasing number of elements in a sequence of rounds with the property that each round enables filtering of writes from future rounds (as was done for MST and BFS). Another is to pre-bucket elements for divide-and-conquer (as was done for convex hull). A third is a technique for partitioning a tree into balanced connected components (as was done for tree contraction). The advantage of these approaches is that they are work-efficient, low-span, and significantly reduce writes. We expect that the techniques will be useful in designing other write-efficient parallel algorithms.

Acknowledgments

This research was supported in part by NSF grants CCF-1314590, CCF-1314633 and CCF-1533858, the Intel Science and Technology Center for Cloud Computing, and the Miller Institute for Basic Research in Science at UC Berkeley.

9. REFERENCES

[1] HP, SanDisk partner on memristor, ReRAM technology. http://www.bit-tech.net/news/hardware/2015/10/09/hp-sandisk-reram-memristor, Oct. 2015.

[2] Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015.

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A prototype phase change memory storage array. In *HotStorage*, 2011.

[4] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. *Technical Report, Tel Aviv University*, 1987.

[5] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Inf. Proc. Letters*, 1990.

[6] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Proc. Letters*, 1979.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, 1998.

[8] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path processing using solid state storage. In *ADMS*, 2012.

[9] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In *ESA*, 2006.

[10] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39, 1996.

[11] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Efficient algorithms under asymmetric read and write costs. *arXiv preprint arXiv:1511.01038*, 2015.

[12] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *SPAA*, 2015.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5), 1999.

[14] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2), 1974.

[15] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simahdri. Write-avoiding algorithms. In *IPDPS*, 2016.

[16] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4), 1996.

[17] T. M. Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning trees. *Inf. Proc. Letters*, 67(6), 1998.

[18] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[19] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO*, 2009.

[20] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *SPAA*, 1996.

[21] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. on Computing*, 21(6), 1992.

[22] X. Dong, N. P. Jouupi, and Y. Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *ICCAD*, 2009.

[23] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *DAC*, 2008.

[24] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and P. Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *SEA*, 2014.

[25] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005.

[26] H. Gazit, G. L. Miller, and S.-H. Teng. *Optimal tree contraction in the EREW model*. Springer, 1988.

[27] M. R. Ghouse and M. T. Goodrich. Fast randomized parallel methods for planar convex hull construction. *Computational Geometry*, 7(4), 1997.

[28] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS*, 1991.

[29] M. T. Goodrich. Finding the convex hull of a sorted point set in parallel. *Inf. Proc. Letters*, 26(4), 1987.

[30] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Proc. Letters*, 1972.

[31] T. Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Graph-Theoretic Concepts in Computer Science*. Springer, 2010.

[32] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, S. Gu, and E. Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014.

[33] www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014.

[34] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[35] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *JACM*, 42(2), 1995.

[36] H. Kim, S. Seshadri, C. L. Dickey, and L. Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *FAST*, 2014.

[37] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2), 1997.

[38] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. on Computing*, 15(1), 1986.

[39] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[40] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 2014.

[41] N. Megiddo. Linear programming in linear time when the dimension is fixed. *JACM*, 31(1), 1984.

[42] G. Miller and J. Reif. Parallel tree contraction and its application. In *FOCS*, 1985.

[43] G. Miller and J. Reif. Parallel tree contraction part 2: Further applications. *SIAM J. on Computing*, 20(6), 1991.

[44] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1), 2010.

[45] H. Park and K. Shim. FAST: flash-aware external sorting for mobile database systems. *J. of Systems and Software*, 82(8), 2009.

[46] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2012.

[47] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(3), 1991.

[48] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *SODA*, 2015.

[49] R. E. Tarjan and U. Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *FOCS*, 1984.

[50] S. D. Viglas. Adapting the B^+-tree for asymmetric I/O. In *ADBIS*, 2012.

[51] S. D. Viglas. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endowment*, 7(5), 2014.

[52] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie. Design implications of memristor-based RRAM cross-point structures. In *DATE*, 2011.

[53] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *ISCAS*, 2007.

[54] Yole Developpement. Emerging non-volatile memory technologies, 2013.

[55] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.

[56] O. Zilberberg, S. Weiss, and S. Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013.