The Read-Only Semi-External Model

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

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Phillip B. Gibbons
Carnegie Mellon University
gibbons@cs.cmu.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Charles McGuffey
Carnegie Mellon University
cmcguffe@cs.cmu.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

We introduce the Read-Only Semi-External (ROSE) Model for the design and analysis of algorithms on large graphs. As in the well-studied semi-external model for graph algorithms, we assume that the vertices but not the edges fit in a small fast (shared) random-access memory, the edges reside in an unbounded (shared) external memory, and transfers between the two memories are done in blocks of size B. A key difference in ROSE, however, is that the external memory can be read from but not written to. This difference is motivated by important practical considerations: because the graph is not modified, a single instance of the graph can be shared among parallel processors and even among unrelated concurrent graph algorithms without synchronization, that instance can be stored compressed without the need for re-compression, the graph can be accessed without cache coherence issues, and the wear-out problems of non-volatile memory, such as Optane NVRAM, can be avoided.

Using ROSE, we analyze parallel algorithms (some existing, some new) for 18 fundamental graph problems. We show that these algorithms are work-efficient, highly parallel, and read the external memory using only a block-friendly (and compression-friendly) primitive: fetch all the edges for a given vertex. Analyzing the maximum times this primitive is called for any vertex yields an (often tight) bound on the (low) I/O cost of our algorithms. We present new, specially-designed ROSE algorithms for triangle counting, FRT trees, and strongly connected components, devising new parallel algorithm techniques for ROSE and beyond.

1 Introduction

Efficient use of the memory hierarchy is crucial to obtaining good performance. For the design and analysis of algorithms, it is often useful to consider simple models of computation that capture the most salient aspects of the memory hierarchy. The *External Memory model* (also known as the I/O or disk-access model) [3], for example, models the memory hierarchy as a bounded internal memory of size M and an unbounded external memory, with transfers between the two done in blocks of size B. The cost of an algorithm is the number of such transfers, called its *I/O complexity*. The model captures the fact that (i) real-world performance is often bottlenecked by the number of transfers (I/Os) to/from the last (slowest, largest) level of the hierarchy used, (ii) that level is used because the second-to-last level is of limited size, and (iii) transfers are done in large blocks (e.g., cache lines or pages). Because of its simplicity and saliency, the External Memory model has proven to be an effective model for algorithm design and analysis [7, 14, 55, 72, 79].

The *Semi-External model* [1] is a well-studied special case of the External Memory model suitable for graph algorithms, in which the vertices of the graph, but not the edges, fit in the internal memory. This model reflects the reality that large real-world graphs tend to have at least an order of magnitude more edges than vertices. Figure 1, for example, shows that *all* the large graphs (at least 1 billion edges) in the SNAP [60], LAW [31] and Azad et al. [9] datasets have an average degree more than 10, and over half have average degree at least 64. The assumptions in the Semi-External model have proven to be effective in both theory and practice [1, 45, 61, 64, 74, 80, 81].

However, the recent emergence of new nonvolatile memory (NVRAM) technologies (e.g., Intel's *Optane DC Persistent Memory*) has added a new twist to memory hierarchies: writes to NVRAM are much more costly than reads in terms of energy, throughput, and wear-out [17, 34, 45, 54, 77, 78]. Neither the External Memory model nor the Semi-External model account for this read-write asymmetry. To partially rectify this, Blelloch et al. [17] introduced the *Asymmetric External Memory model*, a variant of the External Memory model that charges $\omega \gg 1$ for writes to the external memory (the NVRAM), while reads are still unit cost (see also [56]). To our knowledge, the Semi-External setting with asymmetric read-write costs has not been studied. Although one could readily define such a model, graph algorithms provide an opportunity to go beyond just penalizing writes, by *eliminating writes to the external memory altogether!*

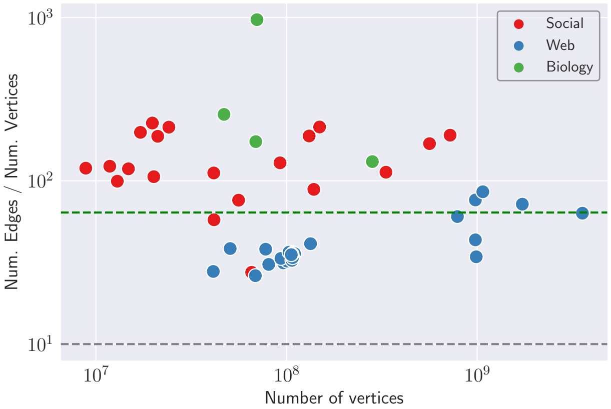

Figure 1: (Adapted from [45]) Number of vertices (n, in log-scale) vs. average degree (m/n, in log-scale) on 52 real-world graphs with $m > 10^9$ edges from the SNAP [60], LAW [31] and Azad et al. [9] datasets. *All* of the graphs have more than 10 times as many edges as vertices (corresponding to the gray dashed line), and 52% of the graphs have at least 64 times as many edges as vertices (corresponding to the green dashed line).

This paper presents the *Read-Only Semi-External (ROSE) model*, for the design and analysis of algorithms on large graphs. As in the Semi-External model, the ROSE model assumes that the vertices but not the edges fit in a small fast (shared) random-access memory, the edges reside in an unbounded (shared) external memory, and transfers between the two memories are done in blocks of size B (where B is the *number of edges* that fit in a block). A key difference in the ROSE model, however, is that *the external memory can be read from but not written to.* The input graph is stored in the read-only external memory, but the output gets written to the read-write internal memory. Unlike general algorithms such as sorting, whose output size is Θ(input size), graph algorithms are amenable to a read-only external memory setting because their output sizes are often $\Theta(n)$ rather than $\Theta(m)$, where n (m) is the number of vertices (edges, respectively) in the graph.

The ROSE model is motivated by practical benefits arising from two main consequences of the model:

No external memory writes: Because of NVRAM's order(s) of magnitude advantage in latency/throughput/wear-out over traditional (NAND Flash) SSDs and in capacity/cost-per-byte over traditional (DRAM) main memory, the emerging setting for large graph algorithms is a hierarchy of DRAM internal memory and NVRAM external memory [45, 50]. In such settings, ROSE algorithms avoid the high performance cost of NVRAM writes. Moreover, ROSE algorithm design is independent of the actual costs of NVRAM writes, which vary depending on access patterns, technologies, and whether the metric of interest is latency, bandwidth, energy, etc. Finally, avoiding writes means avoiding NVRAM wear-out and wear-leveling overheads.

Table 1: Analysis of graph algorithms in ROSE, for a graph G of n vertices and m edges, assuming $m = \Omega(n)$. * denotes that a bound holds in expectation and ‡ denotes that a bound holds with high probability or *whp* ($O(kf(n))$ cost with probability at least $1 - 1/n^k$). † denotes the bound assumes the average degree $d_{avg} = m/n = O(\log n)$; for larger d_{avg}, one of the logs in the depth should be replaced by d_{avg}. B, the block size, is the number of edges that fit in a block. d_G is the diameter of G. r_{src} is the eccentricity from the source vertex. Δ is the maximum degree. α is the arboricity of G. $L = \min(\sqrt{m}, \Delta) + \log^2 \Delta \log n / \log\log n$. ρ is the peeling complexity of G [44]. W_{SP}, D_{SP}, and Q_{SP} are the work, depth, and I/O complexity of a single-source shortest path computation, which depends on the metric used for the FRT trees.

Problem	**Work**	**Depth**	**I/O Complexity**
Triangle Counting	$O(\alpha m)^*$	$O(\alpha \log n + \log^2 n)^\ddagger$	$O(\alpha(n + m/B))$
FRT Trees	$O(W_{SP} \log n)^*$	$O(D_{SP} \log n)^\ddagger$	$O(Q_{SP} \log n)^*$
Strongly Connected Comp	$O(m \log n)^*$	$O(\mathrm{d}_G \log^3 n)^\ddagger$	$O((n + m/B) \log n)^*$
Breadth-First Search	$O(m)$	$O(\mathrm{d}_G \log n)^\dagger$	$O(n + m/B)$
Weighted BFS	$O(r_{src} + m)^*$	$O(\mathrm{r}_{src} \log n)^{\ddagger\dagger}$	$O(n + m/B)$
Bellman-Ford	$O(\mathrm{d}_G m)$	$O(\mathrm{d}_G \log n)^\dagger$	$O(\mathrm{d}_G(n + m/B))$
Single-Source Widest Path	$O(\mathrm{d}_G m)$	$O(\mathrm{r}_{src} \log n)^\dagger$	$O(n + m/B)$
Single-Source Betweenness	$O(m)$	$O(\mathrm{d}_G \log n)^\dagger$	$O(n + m/B)$
$O(k)$**-Spanner**	$O(m)$	$O(k \log n)^{\ddagger\dagger}$	$O(n + m/B)$
LDD	$O(m)$	$O(\log^2 n)^{\ddagger\dagger}$	$O(n + m/B)$
Connectivity	$O(m)^*$	$O(\log^3 n)^{\ddagger\dagger}$	$O(n + m/B)^*$
Spanning Forest	$O(m)^*$	$O(\log^3 n)^{\ddagger\dagger}$	$O(n + m/B)^*$
Graph Coloring	$O(m)$	$O(\log n + L \log \Delta)^{*\dagger}$	$O(n + m/B)$
Maximal Independent Set	$O(m)$	$O(\log^2 n)^{\ddagger\dagger}$	$O(n + m/B)$
Biconnectivity	$O(m)^*$	$O(\mathrm{d}_G \log n + \log^3 n)^{\ddagger\dagger}$	$O(n + m/B)^*$
PageRank Iteration	$O(m)$	$O(\log n)$	$O(m/B)$
k**-core**	$O(m)^*$	$O(\rho \log n)^\ddagger$	$O(n + m/B)$
Apx. Densest Subgraph	$O(m)$	$O(\log^2 n)$	$O(n + m/B)$

A read-only input graph: Because the graph is not modified, a single instance of the graph can be shared among parallel processors and even unrelated concurrent graph algorithms without synchronization. Because data from NVRAM, like DRAM, is brought into CPU caches that are kept coherent by hardware, read-only access means that these cache lines will avoid the costly invalidation that arises with concurrent readers and writers (or concurrent writers). Finally, graphs are often stored in compressed format [44, 70], to reduce their footprint and the memory bandwidth needed to access them. Accessing the graph in a read-only manner still requires runtime decoding, but avoids runtime re-encoding overheads and allows for better (offline, encode-time heavy) compression.

Another twist introduced by NVRAM is that, unlike SSDs, read latency and throughput are only modestly worse than DRAM [54, 77]. Thus, while the I/O complexity (number of external memory reads) remains a good measure, it may no longer be the dominant cost in practice. Accordingly, ROSE includes separate measures for computation *work* and *depth*, standard measures for analyzing parallel algorithms [42, 57].

Algorithm design in the ROSE model. While the benefits of restricting the external memory to be read-only are clear, the question remains as to whether one can design fast and efficient algorithms in the ROSE model. We show that indeed such algorithms exist. Specifically, we analyze parallel algorithms (some existing, some new) for 18 fundamental graph problems. As shown in Table 1, most of the algorithms are work-efficient and highly parallel (often $O(\text{polylog}(n))$ depth). Interestingly, all the algorithms read from the external memory using only a single primitive, FETCHEDGES(v): *Fetch all of the incident edges for a given vertex v.* Because the ROSE model makes the reasonable assumption that the edges for a given vertex are stored consecutively in external memory (optionally compressed), this leads to good I/O complexity. Analyzing the maximum times FETCHEDGES is called for any vertex yields an (often tight) bound on the (low) I/O complexity of our algorithms. Although the bounds (typically, $O(n + m/B)$) may not be optimal for low-degree graphs, many real-world graphs have average degree at least B, in which case the I/O complexity is optimal. In particular, the block size in bytes of Optane DC Persistent Memory is 256. Assuming say 4 bytes per edge, this means 64 edges fit in a block, i.e., $B = 64$ (recall that B is measured in edges not bytes). Figure 1 shows that 52% of the graphs have average degree at least 64.

Our new algorithms are specially-designed ROSE algorithms for triangle counting, FRT trees, and strongly connected components (SCC). Many interesting algorithmic techniques are developed in these algorithms. For triangle counting, our new algorithm requires integrating ideas from the both the classic Chiba-Nishizeki triangle counting algorithm and low out-degree orderings of the vertices to achieve work-efficiency, low depth, and low I/O complexity. For FRT trees, we propose a search-centric view of the algorithm, which is different from all previous algorithms and requires careful algorithm design and analysis. Finally, the ROSE SCC algorithm avoids the edge removal process that is part of the state-of-the-art parallel SCC algorithm [22], while achieving the same work bounds as this existing SCC algorithm. Hence, we believe that the algorithmic insights in this paper may be of interest even in a shared-memory setting without considering NVRAMs and the ROSE setting.

In summary, the two main contributions of this paper are:

- We introduce the Read-Only Semi-External (ROSE) model for the design and analysis of large graph algorithms, motivated by real-world systems considerations.
- We design efficient ROSE algorithms for triangle counting, FRT trees, and SCC, using novel techniques. We also show that 15 existing graph algorithms are efficient in the ROSE model.

2 The ROSE Model

2.1 Model Definition Consider a graph $G(V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Depending on the graph problem being studied, G is (i) undirected or directed, and (ii) weighted or unweighted. We assume that G has neither (undirected) self-edges nor duplicate edges. When reporting bounds, we assume that $m = \Omega(n)$ and indeed semi-external models are relevant only when $m \gg n$. Let $diam(G)$ be the unweighted (hop) diameter of G, r_v be the eccentricity of v or the longest shortest-path distance between v and any vertex u reachable from v, and $deg(v)$ be the degree of vertex v.

The ***Read-Only Semi-External (ROSE)*** model consists of a read-write random-access internal memory of $\Theta(n)$ words and a read-only block-access external memory of unbounded size. Words are $\Theta(\log n)$ bits. Transfers from the external memory to the internal memory (i.e., external memory reads) are done in blocks of size B, where B is measured as the number of edges that fit in a block. The ***I/O complexity*** Q of an algorithm is the number of such transfers. The input graph resides in the external memory and the program output gets stored in the internal memory (thus, the model is restricted to graph problems with $O(n)$ output size). We assume the following canonical form for the input graph layout: vertices are numbered 1 to n and the graph is stored in standard compressed sparse row (CSR) format as consecutive blocks in the external memory. An adversary controls the numbering of the vertices (and hence their ordering in the CSR format). We assume any compression of the graph is done in a manner that enables fast decompression of individual compressed blocks.

The ***work*** W of an algorithm is the total number of instructions using only the internal memory (i.e., not counting external memory reads, which are accounted for in the I/O complexity). For parallel algorithms, we assume the binary-forking model [8, 19, 29], which is widely used in analyzing parallel algorithms [2, 4, 16, 20, 24, 44, 75]. In this model, a running thread can spawn two child threads using a `fork` instruction, and then it resumes only after the children finish. This supports nested-parallelism. All threads share both the internal and external memory. A compare-and-swap (CAS) instruction is allowed on individual words of the internal memory. Some of our algorithms also make use of the fetch-and-add (FA) and priority-write (PW) instructions, which are widely used in the design of parallel algorithms [43, 44, 45, 68]. The ***depth*** D (also called the *span*) of a computation is the length of the longest chain of dependences for instructions. In addition to the $O(n)$ shared-memory accessible to all threads, we assume each thread can allocate $O(\text{polylog}(n))$ memory in a stack fashion (i.e., the memory allocated after a `fork` needs to be freed before the child finishes). A work-stealing scheduler can execute a computation in $W/p + O(D)$ time with high probability on p processors and $O(n + p\,\text{polylog}(n))$ internal memory [12, 30]. In practice, since $p \ll n$, we can ignore the lower-order term

$O(p\,\text{polylog}(n))$, and more discussions for memory allocation and usage can be found in [51]. We seek parallel algorithms that are *work-efficient*, i.e., their work asymptotically matches the best sequential algorithm, and *highly parallel*, i.e., their depth is $\text{polylog}(n)$ or at most $O(diam(G) \cdot \text{polylog}(n))$ (note that most large real-world graphs have small diameter).

2.2 Related Work External Memory and Semi-External models are block-based models, in that they transfer data between the internal and external memory in blocks that hold multiple data words. As noted in Section 1, prior work on such models did not study the case of a read-only external memory. Likewise, other sequential and parallel block-based hierarchical memory models, e.g., the *Ideal Cache* model used in *cache-oblivious* algorithm design [49], the *Multicore-Cache* model [15], and the *Parallel Memory Hierarchy* model [5], do not have read-only memory as part of the model. Carson et al. [35] studied *write-avoiding algorithms*, which seek to minimize the number of external memory writes, but does not disallow them.

For non-block-based models, most prior work accounting for asymmetric read-write costs in the external memory has simply charged more for external memory writes [12, 13, 17, 18, 21, 23, 53]. An exception is our recent work on Sage [45], which defines a cost model that charges $\omega \gg 1$ for external memory writes, but looks to design algorithms that perform no external memory writes. Unlike the present paper, that paper did not formalize a read-only model, did not consider block transfers, allowed more than $O(n)$ internal memory in a key variant, and allowed the input graph to be stored replicated on each socket of a multi-socket machine (to avoid costly NUMA effects). We will show that a number of the Sage algorithms are efficient in the ROSE model (see Section 7).

In the *Semi-Streaming* model [48, 63], graph algorithms can read or write to an internal memory of $O(n \cdot \text{polylog}(n))$ bits and can only read the graph in a sequential streaming order (with possibly multiple passes). This restrictive access to the graph is block-transfer-efficient (m/B transfers per pass), and so any semi-streaming algorithm with W work, t passes, and only $O(n \log n)$ memory bits is also a ROSE algorithm with W work and tm/B transfers. The ROSE model, on the other hand, is not limited to sequential streaming order.

Classic definitions of space complexity assume that the input is in a read-only memory and that the algorithm uses S space if its read-write working space is S bits. Any lower bounds for time in $S = O(n \log n)$ space ($O(n)$ words) immediately carry over to a lower bound for work in the ROSE model. Some classic models allow for an append-only output stream, to enable algorithms whose output size is greater than S. In theory, the ROSE model could be likewise extended. Such *Limited Workspace* models have attracted considerable recent attention in the computational geometry community [10], devising new computational geometry algorithms whose running time is a function of s, the working space in words ($S = s \log n$ bits), $1 \le s \le n$.

To get around stringent lower bounds for read-only models, recent work has studied relaxations that still seek to minimize the additional working space beyond the input graph. If $K = O(m \log n)$ bits are used to store the input graph, *in-place* algorithms [36] are allowed to use those K bits and $\text{polylog}(n)$ more space as the only read-write memory. Gu et al. [51] study in-place graph algorithms in the parallel setting, where sublinear additional space is allowed. *Restore* algorithms [37] are in-place algorithms that must restore the input graph to its original state at the end of the algorithm. As discussed in Section 1, the ROSE model's read-only input graph provides additional benefits beyond just saving space.

Our work is the first hierarchical model to combine the semi-external assumption (vertices fit in internal memory, but edges do not) with a random-access read-only external memory with block transfers. Moreover, unlike most prior work limiting read-write working space, we focus on (efficient) parallel algorithms.

3 The FETCHEDGES Primitive

A key property of the algorithms analyzed in this paper is that they all read the external memory using only a block-friendly (and compression-friendly) primitive, FETCHEDGES(v), that fetches all of the incident edges in the input graph for a vertex v.

Because the edge list of a vertex is contiguous in the external memory, for each block that contains part of that list, at least one of the following must be true: (i) the block contains the beginning of the edge list, (ii) the block contains the end of the edge list, and/or (iii) the block consists entirely of elements of the edge list. For a vertex v, there can be at most one block each of the first and second types, and $deg(v)/B$ blocks of the third type. The call to FETCHEDGES causes one transfer of each of these blocks and no other transfers, resulting in the following lemma.

LEMMA 3.1. *A call to* fetchEdges(v) *causes at most* $\lceil deg(v)/B \rceil + 1$ *transfers from external memory.*

Note that the special case of a **parallel foreach** $(u, v) \in E$ loop to fetch all edges requires only m/B transfers, because it can be implemented using FETCHEDGES on the vertices in CSR order.

We define a ***k*-read ROSE** algorithm to have the following properties: (1) It only reads data from the graph using the FETCHEDGES primitive, and (2) k is an upper bound on the number of times FETCHEDGES(v) is called for any vertex v in the graph. Using Lemma 3.1, we can show that the I/O complexity of a k-read ROSE algorithm is at most $k(\sum_{v \in V}(\lceil deg(v)/B \rceil + 1)) = O(k(n + m/B))$:

Algorithm 1: The ROSE triangle-counting algorithm

```
Input: An undirected graph $G = (V, E)$, and an *ordering*
       over the vertices $<_T$, described in the text below.
Output: The triangle count, $T_G$, of $G$

1  Define $N^+(v) = \{(u, v) \in E \text{ s.t. } u >_T v\}$
2  Set $T_G \leftarrow 0$
3  Set $R \leftarrow V$
4  while $R \neq \emptyset$ do
5      Let $A \leftarrow \{a_1, \ldots, a_c\} \subseteq R$ s.t. $\sum_1^c |N^+(a_i)| \leq 2n$
6      Let $R \leftarrow R \setminus A$
7      Build parallel hash tables, $H_a$ representing
         $N^+(a), \forall a \in A$
8      parallel foreach $(u, v) \in E$ do
9          if $u >_T v$ and $u \in A$ then
10             Let $T_{uv} \leftarrow |N^+(v) \cap N^+(u)|$, computed by
                 hashing $N^+(v)$ into $H_u$
11             Atomically increment $T_G$ by $T_{uv}$
12 return $T_G$
```

THEOREM 3.1. *The I/O complexity of a k-read ROSE algorithm is at most* $O(k(n + m/B))$.

For simplicity, we will only analyze the I/O complexity of uncompressed graphs, but the analysis can be readily adapted to the compressed graph case (with an updated Theorem 3.1).

In the remainder of this paper, we will frequently make use of this theorem as a simple means to derive I/O complexity bounds. As an example, consider the breadth-first search (BFS) algorithm defined in Dhulipala et al. [45]. This algorithm starts at the root and repeatedly expands the frontier of the BFS tree one level at a time in parallel. Each vertex in the frontier use FETCHEDGES to obtain a list of its neighbors, and a conditional check ensures that vertices are not revisited. To ensure that the number of simultaneously fetched edges is $O(n)$ (and not $O(m)$) in the worst case, the algorithm uses a special EDGEMAPCHUNKED function which does not incur additional external memory reads. Thus, the algorithm is a 1-read ROSE algorithm, with I/O complexity $O(n + m/B)$.

4 Triangle Counting

In this section, we present a work-efficient ROSE triangle-counting algorithm whose depth is parametrized in terms of the arboricity of the input graph.

Overview. Our approach in this paper is to parallelize the classic RAM-model triangle-counting algorithm due to Chiba and Nishizeki [38] (the *CN algorithm*) that runs in $O(\alpha m)$ work where α is the arboricity of the input graph, i.e., the minimum number of disjoint forests that the edges of the graph can be decomposed into. The CN algorithm works by intersecting the neighbors $N(u)$ and $N(v)$ for each edge $(u, v) \in E$ by hashing the lower-degree endpoint's neighbors into the higher-degree endpoint's neighbors. They then show the following elegant fact:

$$\sum_{(u,v)\in E} \min(deg(u), deg(v)) = O(\alpha m) \tag{4.1}$$

Because $\alpha \leq \sqrt{m}$, the worst-case running time of this algorithm on the RAM-model is $O(m^{3/2})$. However, for sparser graphs with $\alpha < m^{1/2}$ the work can be significantly better. For example, planar graphs and constant genus graphs have $\alpha = O(1)$, and so the algorithm runs in linear-time on such graphs.

The challenge to overcome in the ROSE model is the fact that the input graph is presented to us in the CSR format, and not as a collection of per-vertex hash tables storing the vertices' neighborhoods. The main idea of our new algorithm is to materialize as many hash tables as will fit in the internal memory and perform partial triangle counting using the materialized neighborhoods. We also make use of a *low out-degree ordering* of the graph in our analysis, which is a total ordering of the vertices such that for each vertex the number of neighbors that come after it in the ordering is $O(\alpha)$. By combining the properties of the CN algorithm and low out-degree orderings with careful use of prefix sums, we obtain the following result:

THEOREM 4.1. *There is a ROSE algorithm for triangle counting with* $O(\alpha m)$ *expected work,* $O(\alpha \log n + \log^2 n)$ *depth* whp, *and* $O(\alpha(n + m/B))$ *I/O complexity.*

Algorithm. We provide the pseudocode for our triangle counting algorithm in Algorithm 1. The algorithm takes as input an undirected graph, and an *ordering* over the vertices $<_T$. The provided ordering does not impact correctness; rather we use it as a tool for analyzing the work and I/O costs of the algorithm, as discussed later in the section. We define $N^+(v)$ to be the neighbors of v ranked higher than v (according to the ordering $>_T$) (Line 1). Our algorithm first initializes the set of vertices to be *removed* (R) to V (Line 3). Then, while R is non-empty, it repeatedly removes a subset of *active* vertices, $A \subseteq R$, from R. (Lines 5–6). The active vertex subset is chosen such that the sum of $|N^+(v)|$ for all vertices $v \in A$ is at most $2n$, and hence fits in the internal memory. Because $|N^+(v)| < n$ for all $v \in V$, we can always find such a subset that sums to more than n, and at most $2n$, with the possible exception of the last loop iteration. A simple way to implement the subset choice is to compute the prefix sum of $|N^+(v)|$ over for all $v \in V$ at the beginning of the algorithm. The algorithm can then keep the total value of $|N^+(v)|$ that is already removed and perform a binary search each round for the vertex with the largest value with difference $\leq 2n$.

For each active subset, the algorithm builds parallel hash tables, H_a, storing the neighbors of the active vertices $a \in A$ (Line 7). It then maps over *all edges* in the graph in parallel and for each edge checks whether the higher degree endpoint of the edge is in A, and thus has its hash table materialized (Line 9). If so, the algorithm computes the intersection size using parallel hashing to hash the vertices in $N^+(v)$ (where v is the lower degree endpoint) into the larger degree vertex's parallel hash table (Line 10). Computing the count can be done using a reduction over the smaller degree vertex's neighborhood. Finally, the algorithm atomically updates the overall triangle count (Line 11).

Although we describe this algorithm using atomics for simplicity, this requirement can easily be removed. The idea is to perform a parallel reduction over all edges in the graph, and then for each edge where the higher-degree endpoint is in A, perform a parallel reduction over the smaller degree's neighborhood to compute the count. Similarly, the set $N^+(v)$ does not actually have to be materialized and can be filtered as the algorithm is iterating through v's neighborhood.

Correctness. We argue that each triangle is counted once. Consider a triangle (u, v, w). Without loss of generality, let $w >_T v >_T u$. Observe that this triangle will be found by the (u, v) edge in the iteration of the while-loop where v is an active vertex ($v \in A$), because $w \in N^+(u)$ and $w \in N^+(v)$. The triangle cannot be found by the (u, w) or (v, w) edge because neither u nor v are present in $N^+(w)$.

Choice of ordering. Our analysis of the work, depth, and I/O complexity of our algorithm relies on a total ordering, $<_T$, of the vertices. We consider two types of orderings, degree ordering and low-outdegree ordering. *Degree* ordering is defined for any two vertices $u, v \in V$ as $u >_{deg} v$ iff $deg(u) > deg(v)$, or $deg(u) = deg(v)$ and $u > v$. *Low out-degree ordering* has the property that for each $v \in V$, the number of v's neighbors that follow it in the ordering is at most $O(\alpha)$. A recent paper by Shi et al. [66] studies a simple parallel algorithm to compute a low out-degree ordering. This algorithm works by removing a constant fraction of the lowest-degree vertices in each round, and adding these vertices to an ordering that is being incrementally constructed. Importantly, each vertex processes its incident edges exactly once in the round when it is peeled, making the algorithm $O(1)$-read. Combining this with the work-depth analysis from Shi et al., we show that in the ROSE model a low-outdegree ordering can be computed in $O(m + n)$ expected work, $O(\log^2 n)$ depth *whp*, and $O(n + m/B)$ I/Os. We refer the interested reader to Shi et al. [66] for more details.

Work and depth. To prove the work and depth bounds in Theorem 4.1, we first bound the number of times the while loop can be invoked. A well-known fact about graphs with arboricity α is that they cannot have many edges: in particular, an arboricity α graph can have at most $O(n\alpha)$ edges. Because each round of the loop (except possibly the last) removes more than n edges, after $O(n\alpha/n) = O(\alpha)$ rounds, R will become empty. On each of these rounds, we process all remaining vertices, and all edges in the graph. The overall work of these steps is thus $O(n\alpha)$ and $O(m\alpha)$, respectively, in the worst case.

To bound the cost of materializing hash tables for vertices when they are active, observe that each vertex constructs a hash table of its incident edges exactly once. The hash table construction can be done in $O(deg(v))$ expected work for each vertex v, and $O(\log n)$ depth *whp*. This is done by using the CAS primitive provided by the model to insert elements into an open-addressed table via linear-probing. The overall expected work of these operations is $O(m)$ across all vertices.

Finally, consider the work of the intersections. Observe that each edge (u, v) is processed in exactly one round, when its higher-ranked endpoint (according to $>_T$) is in A. We call this round the *active round* for (u, v), and assume w.l.o.g. that $u >_T v$. The work of processing this edge is $O(1)$ in the other rounds because we simply scan over it and do nothing. What remains is to bound the work of processing the edge in its active round. There are two cases based on the type of ordering being used by the algorithm.

Degree Ordering. In the active round, we hash $|N^+(v)| \leq deg(v)$ times into H_u. Using Equation 4.1, we show that the total work for processing each edge in its active round is at most $\sum_{(u,v)\in E} \min(deg(u), deg(v)) = O(\alpha m)$. Therefore the total intersection work is $O(\alpha m)$.

Low Out-degree Ordering. In this case, the work can be bounded more directly, since the main property of the low-outdegree ordering is that $|N^+(v)| \leq O(\alpha)$. Therefore, the total work for processing each edge in its active round is $\sum_{(u,v)\in E} O(\alpha) = O(\alpha m)$.

Combining the overall work of each step results in a total algorithmic work of $O(\alpha(m + n))(= O(\alpha m)$ since we assume $m \in \Omega(n))$. For a low out-degree ordering, the overall depth of the algorithm is $O(\alpha \log n + \log^2 n)$, since there are α many rounds that each run a $O(\log n)$ depth reduction over all edges, and the depth of the low-outdegree orientation is $O(\log^2 n)$. For a degree ordering, the depth is just $O(\alpha \log n)$.

I/O complexity. We now argue that Algorithm 1 has low I/O complexity by showing that it is an $O(\alpha)$-read algorithm. First, note that each vertex performs one call to FETCHEDGES in the active round where it materializes its hash table. The algorithm also processes all edges of the graph $O(\alpha)$ times, contributing another $O(\alpha)$ calls to FETCHEDGES per vertex. The remaining calls for each vertex u come from edges (u, v) where $u <_T v$. Specifically, we must bound the maximum out-degree of each vertex in the ordering. For the degree-ordering, unfortunately the maximum out-degree can be up to $O(\sqrt{m})$ in the worst case [71], leading to an $O(\sqrt{m})$-read algorithm. On the other hand, for a low out-degree ordering, the maximum out-degree is $O(\alpha)$. Combining this with our previous observations, we see that the total number of FETCHEDGES calls for each vertex is $O(\alpha)$, leading to an $O(\alpha)$-read algorithm, and an I/O complexity of $O(\alpha(n + m/B))$.

5 Constructing FRT Trees

The FRT tree [47], proposed by Fakcharoenphol, Rao, and Talwar in 2003, is an asymptotically optimal algorithm to generate probabilistic tree embeddings [11], which embed a finite metric (X, d_X) into a distribution of tree metrics with a minimum expected distance distortion. In particular, for every pair of elements $x, y \in X$, the tree distance is always no less than $D_X(x, y)$, and at most $O(D_X(x, y)\log n)$ in expectation. FRT trees have been used in many applications, such as (1) many practical algorithms with good approximation bounds, such as the k-median problem, buy-at-bulk network design [27]; and network congestion minimization [65]; (2) network algorithms including the generalized Steiner forest problem, the minimum routing cost spanning tree problem, and the k-source shortest paths problem [58]; (3) solving symmetric diagonally dominant (SDD) linear systems [41]; (4) construction of approximate distance oracles (ADOs) [25]; and (5) estimating the influence of vertices in a network [40, 46].

In this paper, we consider the input as a graph metric (G, d_G), where $G = (V, E)$ contains n vertices and m edges, and d_G is shortest-path distance. Recent work by Blelloch et al. [22, 25] discussed algorithms for FRT tree construction on a graph both sequentially and in parallel. Both algorithms construct the *least-element (LE) lists* [39] as an intermediate representation, and then construct the FRT tree from the LE-lists using an algorithm by Blelloch, Gupta, and Tangwongsan [27]. Unfortunately, we cannot directly apply this approach here due to the space limitations of the ROSE model—an LE-list require storing an expected $2n\ln n$ vertex indices and distances which are generated from n single-source shortest-paths (SSSP) searches. Thus, theoretically the algorithm does not fit into the ROSE model, and in practice, storing $2\ln n$ numbers per vertex likely precludes storing all LE lists in DRAM when the graph contains tens of billions of vertices. Meanwhile, the FRT tree only has $O(n)$ space that fit in the ROSE model. Therefore, our goal in this paper is to design a new algorithm for constructing FRT trees without explicitly generating the LE-lists. Instead, in our approach we generate the FRT trees by levels, directly based on the distances from the SSSP searches. In the rest of this section, we will first review the existing algorithms, and then present our new approach.

A recent work by Andoni, Stein, and Zhong [6] shows an $\tilde{O}(m)$ work, polylogarithmic depth algorithm to construct FRT trees, but it requires $\tilde{O}(m)$ intermediate space and does not fit into the ROSE model.

5.1 Definitions, Existing Algorithms, and Intuition

We first review the definitions for LE-lists, FRT trees, and existing sequential and parallel algorithms for constructing FRT trees from a graph [22, 25, 27].

LE-lists. Given an ordering of the vertices, the Least-Element lists (LE-lists) for a graph (either unweighted or with non-negative weights) are defined as follows.

DEFINITION 1. (LE-LIST [39]) *Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, the* **LE-lists** *are:*

$$L(v_i) = \left\{\langle v_j, d_G(v_i, v_j)\rangle \mid v_j \in V, d_G(v_i, v_j) < \min_{k=1}^{j-1} d_G(v_i, v_k)\right\}$$

sorted by $d_G(v_i, v_j)$, in decreasing order.

In plain language, a vertex v_j is in vertex v_i's LE-list if and only if there are no earlier vertices ($v_k, k < j$), that are closer to v_i. Often one stores with each vertex v_j in $L(v_i)$ the distance $d_G(v_i, v_j)$. Typically a random ordering of the vertices is used, which ensures the all LE-lists have length $O(\log n)$ *whp*.

Radix-trees. Given an alphabet Σ, and a set of strings S each from Σ^*, a radix-tree[1] of S is generated by taking the trie on S and then removing vertices with one child by combining the incident edges, typically by appending their characters. All interior nodes in a radix-tree therefore have at least two children, and hence the total number of nodes is $O(|S|)$.

FRT trees and Compressed FRT Trees. The FRT algorithm is based on a random permutation of the input points, and a parameter $\beta \in [1, 2)$ randomly selected from the probability density function $f_b(x) = 1/(x \ln 2)$. We assume that the weights are normalized so that $1 \le d_X(x, y) \le \Delta = 2^{\bar{\delta}}$ for all $x \neq y$, where $\bar{\delta}$ is a positive integer. The original algorithm [47] was described as a top-down clustering algorithm, generating a laminar family of clusters. This corresponds to a tree in which the edge weights start at Δ at the root and at each level decrease by a factor of two going down the tree. Such a tree, however, can have a number of nodes that is at least quadratic in the input size. Therefore, in this paper we build a compressed FRT tree [28], for which nodes in the FRT tree with a single child are spliced out and the incident edge weights combined. This transformation preserves distances in the tree. The leaves correspond to the input points, and since all internal nodes have at least two children, the tree is of size $O(n)$. The tree also has depth $O(\log n)$ *whp* [28].

Compressed FRT Trees from LE-lists. The compressed FRT tree can be generated from LE-lists directly [25, 28], avoiding the large number of nodes in the full FRT tree. This can be done in three steps:

1. Generate the LE-list for each point based on the random permutation of the input. Each such list has size $O(\log n)$ *whp*.

[1] Also called PATRICIA trees or radix tries.

2. Take all of the distances $d_G(v_i, v_j)$ in the LE-lists and replace them with rounded log-distances $\lfloor \log_2 \frac{\beta\cdot\Delta}{d_G(v_i,v_j)} \rfloor$, and then only keep the first entry among equal distances.

3. Treating each modified LE-list as a string, where each character is a (vertex, log-distance) pair, build a radix tree on all the lists. Weight the edges based on the top "character" on the combined edge.[2]

Unfortunately, this algorithm will not suffice for our purposes since our goal is to use only $O(n)$ space, while the LE-lists themselves require $O(n \log n)$ space. As mentioned, we plan to integrate the generation of LE-lists and generation of the tree. This requires understanding and adapting the parallel LE-list algorithm. Also the previous parallel algorithm using this idea [28] requires $O(n \log^2 n)$ work for the third step since it requires a lexicographic sort. Our new algorithm also improves this bound to $O(n \log n)$ work, which might be of independent interest to the ROSE model.

Generating LE-lists in Parallel. We start with the BGSS parallel algorithm for constructing LE-lists (Algorithm 2) [22]. The algorithm runs for $\log n$ rounds, where each round doubles the number of vertices from which it does SSSP searches "in parallel"—i.e., on the r'th round it runs SSSP searches from the next 2^{r-1} vertices. The set S_i captures all vertices that are closer to the i'th vertex than earlier vertices (the previous closest distance is stored in $\delta(\cdot)$). Line 4 computes S_i using a single-source shortest paths (SSSP) algorithm (e.g., Dijkstra's algorithm or a different shortest path algorithm [26, 59, 73, 76]). Then the $\delta(\cdot)$ values are updated based on the searches in this round. This algorithm requires $O(W_{SP}(n,m) \log n)$ expected work and $O(D_{SP}(n,m) \log n)$ depth [22] *whp*, where $W_{SP}(n,m)$ and $D_{SP}(n,m)$ are the work and depth, respectively, to compute SSSP for a graph with n vertices and m edges. We also care about the IO-complexity $I_{SP}(n,m)$ to compute the SSSP. We note that for unweighted graphs, we can use BFS giving $O(m)$ work, $O(\mathrm{d}_G \log n)$ depth, and $O(n + m/B)$ IO-complexity, where d_G is the diameter of the graph. Otherwise we could use a variety of solutions [26, 59, 73, 76], although they would have to be analyzed on the ROSE model.

A key observation, for our purposes, is that after each round the algorithm has generated a prefix of each LE-list. In particular, after round r, for each vertex $i \in \{1, \ldots, n\}$, the LE-list for i will consist of its LE-list with all entries for vertices with indices up to 2^r. Each round extends each prefix by an expected constant number of additional elements. We will use this observation to extend the FRT tree on each round without keeping the full LE-lists. This will be done by keeping a kind of "prefix" of the compressed FRT tree that is updated on each round by adding appropriate descendants to the tree from the previous round.

Algorithm 2: The BGSS algorithm for constructing LE-lists in parallel [22].

```
Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: The LE-lists $L(\cdot)$ of $G$
1  Set $\delta(v) \leftarrow +\infty$ and $L(v) \leftarrow \varnothing$ for all $v \in V$
2  for $r \leftarrow 1$ to $\log_2 n$ do
3      parallel foreach $i \in \{2^{r-1}, \ldots, 2^r - 1\}$ do
4          Let $S_i = \{u \in V \mid d(v_i, u) < \delta(u)\}$
5      parallel foreach $u \in \bigcup_i S_i$ do
6          Let $l(u) \leftarrow \{v_i \mid u \in S_i\}$
7          Sort $l(u)$ based on $d_G(v_i, u)$ in descending order
             and filter out $v_i$ that are not in ascending order
8          Append $l(u)$ to the end of $L(u)$
9          $\delta(u) \leftarrow d(v_j, u) \mid v_j$ is the last element in $l(u)$
10 return $L(\cdot)$
```

5.2 Our Algorithm

We now describe a parallel work-efficient algorithm that only requires $O(n)$ temporary space, and thus can be implemented in the ROSE model.

THEOREM 5.1. *A compressed FRT tree can be built from a metric based on a graph with n vertices and m edges using $O(W_{SP}(n,m) \log n)$ expected work, $O(D_{SP}(n,m) \log^2 n)$ depth* whp, *and $O(I_{SP}(n,m) \log n)$ expected IO-complexity where $W_{SP}(n,m)$, $D_{SP}(n,m)$, and $I_{SP}(n,m)$ are work, depth and IO-complexity to compute SSSP, using $O(n)$ auxiliary space.*

There are two main algorithmic improvement in the new ROSE algorithm, and we first overview the high-level ideas and then go into details. The first insight is to avoid constructing the entire LE-lists $L(\cdot)$. We note that the LE-lists have $O(n \log n)$ elements *whp*, and the outer loop for r in line 2 in Algorithm 2 also runs for $O(\log n)$ iterations. In expectation, each iteration will search out for $O(n)$ vertices to add to the LE-lists. Hence, if we can directly integrate the search values of S_i to the FRT tree, rather than wait until all elements in LE-lists $L(\cdot)$ are computed, then we can bound the memory usage to be linear (some care needs to be taken since the $O(n)$ is only in expectation). The second insight is that, as mentioned above, the existing parallel algorithm [28] is not work optimal due to sorting the LE-lists lexicographically. We observe that the FRT tree does not need an order, either for the leaf nodes or the interior nodes, since when querying two vertices which correspond to two leaf nodes in the FRT tree, the output is the tree-path distance, and the ordering of the tree nodes does not matter.

An algorithmic overview. The pseudocode of the new ROSE FRT algorithm is given in Algorithm 3. It runs for $O(\log n)$ rounds (line 5), and in each round, we not only apply the

[2] This could cause distances to differ from the original FRT tree by a constant factor, but with more care the distances can be made identical.

Algorithm 3: The ROSE FRT algorithm

```
Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: An FRT tree $T_{\pi,\beta}$
1  Create a uniformly random permutation $\pi : V \rightarrow V$ of the
     vertices in $G$.
2  Pick $\beta \in [1, 2]$ using the probability density function
     $f_B(x) = 1/(x \ln 2)$.
3  We will maintain the following across rounds:
     1. A partial FRT tree (i.e., the top part), which we will build
        from the root node into the full tree.
4    2. For each $v \in V$, a pointer $\tau_v$ to the leaf of the partial FRT
        tree that will eventually contain $v$. All $\tau_v$ initially point
        to the root.
     3. For each $v \in V$, the smallest distance to $v$ from vertices
        processed so far, $\delta(v)$. All $\delta(v)$ are initially set to $+\infty$.
5  for $r \leftarrow 1$ to $\log_2 n$ do
6      parallel foreach $i \in \{2^{r-1}, \ldots, 2^r - 1\}$ do
7          Let $S_i = \{u \in V \mid d(v_i, u) < \delta(u)\}$
8      parallel foreach $u \in V$ do
9          Let $l(u) \leftarrow \{\langle v_i, \rho_i = \lfloor \log_2 \frac{\beta\Delta}{d_G(v_i,u)} \rfloor\rangle : 1 \le i \le$
             $n \mid u \in S_i\}$
10         Sort $l(u)$ based on $\rho_i$ in descending order and filter out
             $v_i$ that are not in ascending order
11     Now collect vertices together using $\langle \tau_u, l(u)\rangle$, and just keep
         one of each equivalence class (duplicates removed), and
         call the resulting set $\bar{S}$ (can be done with a semisort)
12     parallel foreach $u \in \bar{S}$ do
13         Let $l'_{(j)}(u) \leftarrow$
             $\{\langle v_{u_1}, \rho_{u_1}\rangle, \ldots, \langle v_{u_{j-1}}, \rho_{u_{j-1}}\rangle, v_{u_j}\} \mid 1 \le j \le |l(u)|$,
             $l''_{(j)}(u) \leftarrow \{\langle v_{u_1}, \rho_{u_1}\rangle, \ldots, \langle v_{u_j}, \rho_{u_j}\rangle\} \mid 0 \le j \le |l(u)|$
14     For all $u \in \bar{S}, j \le |l(u)|$, using a semisort collect together
         based on key $\langle \tau_u, l'_{(j)}(u)\rangle$ along with value $\rho_{u_j}$, and count
         the appearance of the keys $\langle \tau_u, l'_{(j)}(u)\rangle$ and $\langle \tau_u, l''_{(j)}(u)\rangle$
         (also using semisort)
15     parallel foreach key $\langle \tau_u, l'_{(j)}(u)\rangle$ after semisort do
16         Sort the value $\rho_{u_j}$ in increasing order
17         if $\tau_u$ has children or $|\langle \tau_u, l'_{(j)}(u)\rangle| < |\langle \tau_u, l''_{(j-1)}(u)\rangle|$
             then
18             Create a tree node for the first entry
19         Create a tree node for every entry but the first one,
             point the root of each node to the predecessor
20     parallel foreach $u \in \bigcup_i S_i$ do
21         Update $\tau_u$ to the current corresponding node
```

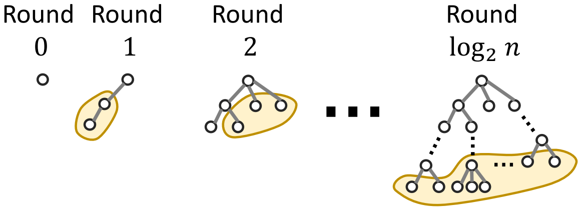

Figure 2: An illustration for the new algorithm to construct FRT trees. We grow the tree a constant number of levels in expectation, and repeat for $\log_2 n$ rounds.

SSSP searches (line 7), but also directly integrate the search results in S_i to the FRT tree and discard S_i after the round. In expectation, the SSSP search result in each round (i.e., $\sum |S_i|$) has size $O(n)$, and later we will discuss how to deal the case when it is $\omega(n)$ in rare cases. When generating the FRT tree, we only create the tree nodes that will eventually show up in the final tree, and the final tree has no more than $2n-1$ tree nodes. All other intermediate steps use space proportional to the size of SSSP search ($\sum |S_i|$), so we can restrict the auxiliary space to be $O(n)$.

The conversion from the SSSP search results to an FRT tree takes $O(n \log n)$ expected work and $O(\log^2 n)$ depth, which is interesting even without considering the ROSE model. The depth can be further improved to $O(\log n)$ if we first do all searches in $O(\log n)$ rounds (line 5), and then run the rest of the algorithm (line 8–21) in one pass, although it then requires $O(n \log n)$ space. The algorithmic insight is that previous algorithms either use a *point-centric view* [22, 25, 27] (for a specific vertex v, we consider which SSSP searches can reach v), or a *level-based view* [47] (consider the partition with different search radii and refine the partition). The new algorithm uses the *search-centric view*: for a specific SSSP search from v_i, we check the reach set and see what tree node this search creates. More specifically, the SSSP search from v_i will reach the vertex set S_i, and we consider and process the FRT tree nodes incident to S_i.

The key idea of the algorithm is to maintain on each round a partial compressed FRT tree (the top part), and for each vertex u that has not been added yet, we keep a pointer τ_u to a tree node that u will eventually be descendant of. In particular, if we have processed points up to i, the compressed FRT tree will include all edges labeled with vertices up to i (recall the edges are created from the LE-lists, and contain a vertex and a distance to that vertex). For instance, in Figure 2, after round 1, we generate the tree nodes incident to the search from v_1. Every other vertex v_i is reached by the search from v_1, and diverges at level $\lfloor \log_2 \frac{\beta \cdot \Delta}{d_G(v_1, v_i)} \rfloor$. We create all nodes at these levels (line 13), and distribute τ_{v_i} for each v_i to the corresponding level. As shown in Figure 2, we repeat this process for $\log_2 n$ rounds, and generate the FRT tree.

Another interpretation of this algorithm is that, on each round the BGSS algorithm generates extensions to the LE-lists using SSSP searches from a new set of points. We apply the same strategy here, but now immediately insert the elements into the partial FRT tree. Roughly speaking, this is done by grouping the extensions for each vertex u by τ_u (the current node pointed to by u in the partial FRT tree), and building the part of the tree that extends that leaf, and then updating all vertices to point to their new τ_u, which will be a descendant of the old one. Note that to do this we need to group by the pair consisting of τ_u and the contents of the the extensions of the LE-lists. This can be done with a semisort since the ordering does not matter.

In the rest of this section, we first describe the algorithm in more detail, prove the correctness, and analyze the cost bounds. Then we discuss how to guarantee the space usage to be $O(n)$ so that it works in the ROSE model. Putting all pieces together, we achieve the bounds in Theorem 5.1.

The search-centric conversion algorithm. We now present our algorithm to convert from the SSSP search results to an FRT tree in $O(n \log n)$ expected work and $O(\log n)$ depth. The algorithm uses the search-centric view and uses semisort as the crucial building block. Recall that semisort takes the n key-value pairs as input, and group all pairs with the same key using linear expected work and $O(\log n)$ depth *whp* [19, 52].

To do so, we borrow the concepts of a partition sequence [27] to better illustrate our algorithm. Given a permutation π and a parameter β, the ***partition sequence*** of a vertex u, denoted by $\sigma^{(u)}_{\pi,\beta}$, is the sequence $\sigma^{(u)}_{\pi,\beta}(i) = \min\{\pi(v) \mid v \in V, d_G(u,v) \le \beta \cdot 2^{\delta-i}\}$ for $i = 0, \ldots, \bar{\delta}$, i.e. point v has the highest priority among vertices up to level i. Then the FRT tree is just a radix tree for the trie constructed based on partition sequence for each vertex. We now explain how the searches in each round modify the partition sequence and therefore the FRT tree nodes.

Observation 1. *Vertex v_i creates the node when there exists two nodes v_j and v_k and the partition sequences of v_j and v_k diverges at v_i.*

More accurately, $\sigma^{(v_j)}$ and $\sigma^{(v_k)}$ have the same prefix, and they diverge at level $x+1$, i.e., $\sigma^{(v_j)}(x) = \sigma^{(v_k)}(x) = v_i$, and $\sigma^{(v_j)}(x+1) \neq \sigma^{(v_k)}(x+1)$ (More details are shown in [27]). We now show Algorithm 3 creates all FRT tree nodes correctly, and does not create more nodes (otherwise the tree size can be larger than $O(n)$ and does not fit in the ROSE model). We first see what FRT tree nodes v_i can create.

Observation 2. *Vertex v_i can only create nodes below the nodes created by v_j when $j < i$, because of the definition of partition sequence.*

Namely, the FRT tree can be generated incrementally, by adding the leaf nodes created by $v_1, v_2, \ldots, v_n$. Recall the case $\sigma^{(v_j)}$ and $\sigma^{(v_k)}$ diverge at level $x+1$. There are two reasons that a FRT tree node is generated: (1) $\sigma^{(v_j)}(x+1) = \sigma^{(v_j)}(x) \neq \sigma^{(v_k)}(x+1)$ (or symmetric), and (2) $\sigma^{(v_j)}(x+1) \neq \sigma^{(v_j)}(x) \neq \sigma^{(v_k)}(x+1)$. For the first case, let $v_i = \sigma^{(v_j)}(x+1) = \sigma^{(v_j)}(x) \neq \sigma^{(v_k)}(x+1)$. Since $\sigma^{(v_j)}$ and $\sigma^{(v_k)}$ diverge at level $x+1$, we must have $\tau_{v_j} = \tau_{v_k}$ and they share a common prefix before the first appearance of v_i, and for v_i's search, they have different σ_i. Hence, in the semisort in line 14, they share the same key (τ_{v_j}) but have different values, so the tree node will be created either in line 18 or line 19. For the second case, suppose $\sigma^{(v_j)}(x) = v_a$, $\sigma^{(v_j)}(x+1) = v_b$, and $\sigma^{(v_k)}(x+1) = v_c$. Based on Observation 2, we must have $a < b$ and $a < c$. By the time when the algorithm searches from v_b and v_c, the parent node is already created by v_a or v_a's ancestor. WLOG, suppose that v_b's search occurs in an earlier round. Then in v_b's search, v_c will stay in the parent node, so the condition $|\langle \tau_u, l'_{(j)}(u)\rangle| < |\langle \tau_u, l''_{(j-1)}(u)\rangle|$ in line 17 is always true and the algorithm will generate a new node for v_j. Then in v_c's search, we know the parent node for v_a has at least one child already, so again the condition in line 17 is satisfied, and a new node for v_k will be generated. If v_b and v_c search in the same round, then both will be the first case, and a new tree node for each vertex will be constructed in line 18.

We have shown that all the tree nodes will be created. We now show that no additional tree node is created. First, each node other than the leaf nodes have at least two children so there is no uncompressed nodes. This is because nodes created in line 19 has one child already (other than the last one which is a leaf), and at least another node will be filled in line 18. For nodes created in line 18, they are either not the first child of the parent node, or the condition $|\langle \tau_u, l'_{(j)}(u)\rangle| < |\langle \tau_u, l''_{(j-1)}(u)\rangle|$ guarantees that at least a later child node will be filled in later. Second, the nodes created in line 18 and line 19 are due to either of the two cases discussed in the previous paragraph, so every node created by Algorithm 3 is necessary.

Fitting in the ROSE model. We have shown how to convert the search results S_i to FRT tree nodes. We know that $\sum_{i=1}^{n} |S_i| = O(n \log n)$ *whp* and there are $\log_2 n$ rounds in the algorithm, and in expectation the overall search size is $O(n)$ [22]. It is easy to check that all steps in line 8 to line 21 use space proportional to $\sum |S_i|$ in one round. If we have cn regular memory size for a reasonable large constant c, it is likely that the algorithm will just run well. However, it is possible that $\sum |S_i|$ is large (i.e., $\omega(n)$) in one round, and of course our algorithm should deal with it rather than crashing.

The solution is to dynamically adjust the batch size in line 6. We assume we have a budget for the regular memory size that can hold cn elements for some constant $c \ge 1$. Once the SSSP searches in one round (line 7) exceed this size, we stop and shrink the range by a half, and repeat if necessary. This will lead to additional work when the resize is triggered, but it will not affect the work asymptotically. This is because in for v_i's search in round r, $\mathbb{E}[|S_i|] = n/2^{r-1}$. When each resizing is triggered, we divide the range into two equal-size halves and each side has $O(n)$ SSSP search size in expectation. Therefore, the failed SSSP searches that reach $O(n)$ vertices is asymptotically bounded.

6 Strongly Connected Components

In this section, we discuss the ROSE SCC algorithm that can achieve the same asymptotic work and slightly larger

```
Algorithm 4: The ROSE SCC algorithm
  Input: A directed graph $G = (V, E)$ with
         $V = \{v_1, \ldots, v_n\}$.
  Output: The set of strongly connected components of $G$.
1  //REACHABILITY in line 7 and 8 for $v_i$ only searches
     vertices $v_j$ such that $M_i = M_j$
2  $M \leftarrow \{(0, \varnothing), (0, \varnothing), \ldots, (0, \varnothing)\}$
3  $S_{scc}, V_{scc} \leftarrow \{\}$
4  for $r \leftarrow 1$ to $\log_2 n$ do
5      parallel foreach $i \in \{2^{r-1}, \ldots, 2^r - 1\}$ do
6          if $v_i \in V_{scc}$ then Break;
7          $S_i^+ \leftarrow$ FORWARD-REACHABILITY($v_i$)
8          $S_i^- \leftarrow$ BACKWARD-REACHABILITY($v_i$)
9          Generate key-value pairs $(v^+, (v_i, +)) \mid v^+ \in S_i^+$
             and $(v^-, (v_i, -)) \mid v^- \in S_i^-$
10     Semisort all the pairs, and let $L_i$ be the set of values
         for pairs with keys $v_i$
11     parallel foreach $i \in \{2^{r-1}, \ldots, 2^r - 1\}$ do
12         if not $(\{(v_j, +), (v_j, -)\} \subseteq L_i), j < i$ then
13             $S_{scc} \leftarrow S_{scc} \cup \{S_i^+ \cap S_i^-\}$
14             $V_{scc} \leftarrow V_{scc} \cup (S_i^+ \cap S_i^-)$
15     parallel foreach $L_i \mid v_i \in V \backslash V_{scc}, L_i \neq \{\}$ do
16         Sort $L_i$ based on the label of the vertices in
             increasing order
17         $(v_l, s_l) \leftarrow L_i(1)$
18         for $j \leftarrow 2$ to $|L_i|$ do
19             $(v_c, s_c) \leftarrow L_i(j)$
20             if $v_c$ is reachable from $v_l$ in $s_l$ direction then
                 $(v_l, s_l) \leftarrow (v_c, s_c)$ ;
21         $M_i \leftarrow (v_l, s_l)$
22 return $S_{scc}$
```

depth as compared to the best existing parallel SCC algorithm without the semi-external constraint. The best existing parallel SCC algorithm is referred to as the BGSS algorithm [22], which takes $O(W_R(n, m) \log n)$ expected work and has $O(D_R(n, m) \log^2 n)$ depth *whp* on an input graph with n vertices and m edges. $W_R(n, m)$ and $D_R(n, m)$ are the work and span, respectively, for a reachability algorithm from a single vertex that visits n vertices, with m out-edges from those vertices. However, this algorithm requires $O(m)$ auxiliary space since it explicitly removes edges during the execution of the algorithm. The high-level idea in the ROSE algorithm is to avoid this edge removal process.

Before we go into the details of the new ROSE SCC algorithm, let's first review the BGSS algorithm that the new algorithm is based on. At the beginning, all vertices are uniformly randomly permuted. BGSS runs in rounds, and in round r, it applies 2^{r-1} forward reachability queries and 2^{r-1} backward reachability queries for vertices with indices $2^{r-1}, \ldots, 2^r - 1$. These reachability searches find the SCCs that these vertices belong to, and cuts edges to partition vertices into disjoint subsets. In particular any edge is removed if any of the reachability queries visited one of its endpoints but not the other. All SCCs that the vertices belong to are removed, and the remaining partitioned graph is left for the next round. The algorithm iterates for $\log_2 n$ rounds and finds all SCCs of a graph.

Since BGSS removes edges explicitly for deciding the vertex subsets, it is not in the ROSE model. The idea in the ROSE algorithm is to give each vertex a label, such that vertices with the same label are in the same partition. The algorithm is correct as long as the partitions defined by the labels are equivalent to the partitions in the original BGSS. For analyzing the partitions, we actually consider a slight variant of the BGSS algorithm in which the vertices are searched (forward and backwards with cutting) in sequential order within a round. The partitions for such a variant at the end of a round are actually those analyzed for the BGSS algorithm [22], and the paper shows that the parallel variants can only be more agressive at partitioning. We will label every vertex with the last forward and last backward search in the round that visited it if run sequentially. With this labeling, edges are cut exactly when the labels for either direction differ on the two end points. This is because different labels imply a search visited one endpoint but not the other, and equal labels implies the last search, and hence all previous searches on either, visited both.

Assume that vertices x, y, and z that are in separate SCCs. Let's consider the following case in one parallel round: x and y can reach z is the forward direction, and the search order is first x then y within this round, and z in a future round. In sequential BGSS, the search from y will reach z iff y is reachable from x, otherwise x's search will disconnect (separate) y and z before y's search. We will take advantage of this property to generate our sequential labels even though we run the searches in parallel. In particular we can look at all searches that reach a vertex z in the parallel (batch) version, scan through those in sequential order (there are only a constant number in expectation and logarithmic *whp*), and determine which would have been the last to visit x in the sequential version.

The ROSE algorithm is described in Algorithm 4. The labels are stored in the M array, and the algorithm runs the rounds in parallel as in BGSS. For all rounds and for all vertices reached in reachability searches in a round, we create a visited-source pair (line 9) and semisort by visited (line 10). More precisely, for each vertex v_i, we collect L_i, the indices of all searches that have $v_i$in their reached set. Now for vertices in L_i, we sort by source index (line 16), and set the temporary label (v_l, s_l) to the earliest (line 17). We now iterate over the rest in increasing order for each search index v_c. If v_c is reachable from v_l's search in s_l direction, then set the current label to (v_c, s_c) (line 20); otherwise leave it.

Whenever we do not change the label, this corresponds to a visit that happened in the parallel algorithm that would not happen in the sequential one (v_l's search on s_l direction would have separated them). We note that v_i can be reached by v_l in at most one direction. Otherwise, v_i and v_l are strongly connected and v_i is removed from V_{scc} already in line 14. Hence, there will be no duplicates with the save vertex in L_i. After generating the final label, we update M_i (line 21).

THEOREM 6.1. *The ROSE SCC algorithm requires $O(W_R(n,m)\log n)$ expected work and has $O(D_R(n,m)\log^2 n)$ depth* whp*, and uses $O(I_r(n,m)\log n)$ I/Os, where $W_R(n,m)$, $D_R(n,m)$, and $I_R(n,m)$ are work, span and I/O-complexity to compute reachability, using $O(n)$ auxiliary space.*

Proof. Similar to the ROSE FRT algorithm, if the searches for all rounds reach $O(n)$ vertices, ROSE SCC does not apply additional reachability searches as the sequential BGSS, which gives to the work and I/O bounds. The additional steps are on computing the vertex labels, with size $O(n)$, so they are in the internal memory. Semisorting in line 10 takes linear expected work and logarithmic depth *whp*. Sorting the L_i lists has the same cost as sorting $l(u)$ in ROSE FRT, which also takes linear expected work and logarithmic depth *whp*. Since *whp* all the reachability searches will touch $O(n\log n)$ vertices in total [22], all the additional work in ROSE SCC is hidden by the cost of reachability searches. Regarding the small possibility that a search for a round in ROSE SCC reaches $\omega(n)$ vertices, we can trigger resizing similarly to ROSE FRT, which will not affect work asymptotically, but the depth of ROSE is increased by a logarithmic factor as compared to BGGS SCC. □

Using BFS for reachability the costs are $O(m\log n)$ expected work, $O(\mathrm{d}_G\log^3 n)$ span *whp*, and $O((n{+}m/B)\log n)$ expected I/O-complexity, where d_G is the diameter of the graph. This is the bounds shown in Table 1.

7 Existing Algorithms in ROSE

In this paper, we also consider a set of 15 parallel graph algorithms recently designed in our previous work on Sage [45]. We will show that each of these algorithms (other than Bellman-Ford) is a $O(1)$-read ROSE algorithm (Section 3), and thus by Theorem 3.1 has low I/O complexity (at worst $O(n+m/B)$) on ROSE. The algorithms inherit the work and depth bounds from [45]. Table 1 (below the mid-line) summarizes the results. As shown in the table, nearly all the algorithms achieve our goals of being work-efficient, highly parallel, and low I/O complexity.

In what follows, we briefly summarize how to show that these algorithms are $O(1)$-read ROSE algorithms. We refer the interested reader to [44, 45] for more details on these algorithms.

Shortest Path Problems We consider six shortest-path problems: *breadth-first search (BFS)*, *integral-weight SSSP (wBFS)*, *general-weight SSSP (Bellman-Ford)*, *single-source betweenness centrality*, *single-source widest path*, and $O(k)$-*spanner*. First, we observe that BFS, wBFS, single-source betweenness centrality, and $O(k)$-spanner all process each vertex at most a constant number of times in their operations. For example, for BFS and wBFS, a vertex v is processed at most once, when the (weighted) breadth-first search frontier contains it. Similarly, in single-source betweenness centrality, a vertex is processed at most twice: once in the forward pass which computes the number of shortest-paths to each vertex, and once in the backwards pass which computes dependency scores [32, 67]. Finally, $O(k)$-spanner works by computing an LDD, which we discuss below, and mapping over the edges incident to all vertices in parallel, and is thus also an $O(1)$-read ROSE algorithm. For Bellman-Ford, note that in the worst case the algorithm can process a vertex $diam(G)$ many times, and thus it is an $O(diam(G))$-read ROSE algorithm. Single-source widest path can be implemented either using an approach similar to wBFS, or Bellman-Ford; the bounds shown in Table 1 show the bounds for the Bellman-Ford based implementation. We note that the wBFS, Bellman-Ford, and single-source widest path algorithms all use the PW primitive.

Connectivity Problems We consider four connectivity problems: *low-diameter decomposition (LDD)*, *connectivity*, *spanning forest*, and *biconnectivity*. The LDD algorithm works similarly to BFS, loading the edges incident to a vertex only in the round where it is processed either as an LDD cluster center, or as a vertex on the boundary of an LDD cluster [45, 62]. Since each vertex is processed exactly once, it is an $O(1)$-read algorithm. We also consider several connectivity algorithms that build on LDD, including the $O(k)$-spanner algorithm described above. Our connectivity and spanning forest algorithms are based on the algorithm by Shun et al. [69], and our biconnectivity algorithm is from Dhulipala et al. [44]. All three algorithms use the modifications described in Dhulipala et al. [45] to run in $O(n)$ space *whp*. We observe that all three of these algorithms only process the edges incident to each vertex in the original graph a constant number of times *whp*, and are thus expected $O(1)$-read ROSE algorithms. We note that the biconnectivity algorithm uses the FA primitive when performing leaffix and rootfix scans, and the SCC algorithm uses the PW primitive [44].

Covering Problems We consider two covering problems: *maximal independent set (MIS)* and *graph coloring*. As shown in Dhulipala et al. [45] both algorithms only use $O(n)$ words of memory. Here, we observe that both algorithms only process the edges incident to each vertex once. For MIS, a vertex is processed either when it is added to the MIS by the algorithm, or in the round where it is removed by one of its neighbors joining the MIS. Similarly, for coloring, a vertex is processed only in the round where it is ready to be colored. Thus, both algorithms are $O(1)$-read ROSE algorithms. We note that the MIS and graph coloring algorithms use the FA primitive.

Substructure Problems We consider two substructure-based problems from prior work: *k-core* and *approximate densest subgraph*. Dhulipala et al. [45] previously argued how both algorithms can be implemented using only $O(n)$ words of internal memory. Here, we observe that their algorithms are actually $O(1)$-read ROSE algorithms. Specifically, for both algorithms, the algorithm processes the edges incident to a vertex exactly once, in the round when the vertex is peeled. We note that both algorithms use the FA primitive.

Eigenvector Problems Lastly, we consider the problem of computing the *PageRank* vector of the graph. Our algorithm is based on the classic PageRank algorithm [33], and is based on the implementation by Dhulipala et al. [45]. Here, we observe that this algorithm processes all of the edges in the graph in every iteration, leading to $O(m/B)$ I/O complexity per-iteration.

8 Conclusion

We have introduced the Read-Only Semi-External (ROSE) Model for graph algorithms. We have analyzed 18 parallel algorithms in this model, and have shown that they are work-efficient, highly parallel, and have strong I/O bounds. Our algorithms make use of the FETCHEDGES primitive to traverse neighbors of vertices, and by analyzing the number of times this primitive is called, we are able to obtain strong I/O bounds for the algorithms in the ROSE model. Finally, our algorithms for triangle counting, FRT trees, and strongly connected components are specially designed for the ROSE model, with novel techniques.

Acknowledgements

This research was supported by DOE Early Career Award DE-SC0018947, NSF CAREER Award CCF-1845763, NSF grants CCF-1725663, CCF-1910030, CCF-1919223 and CCF-2028949, Google Faculty Research Award, VMware University Research Fund Award, the Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma), DARPA SDH Award HR0011-18-3-0007, and the Applications Driving Algorithms (ADA) Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, Mar 2002.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theoretical Computer Science (TCS)*, 35(3):321–347, 2002.

[3] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[4] Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 84–95, 2014.

[5] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, pages 116–123, 1993.

[6] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *ACM Symposium on Theory of Computing (STOC)*, pages 322–335, 2020.

[7] Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2010.

[8] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2):115–144, Apr 2001.

[9] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. HipMCL: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 46(6):e33–e33, 2018.

[10] Bahareh Banyassady, Matias Korman, and Wolfgang Mulzer. Computational geometry column 67. *SIGACT News*, 49(2), 2018.

[11] Yair Bartal. On approximating arbitrary metrices by tree metrics. In *ACM Symposium on Theory of Computing (STOC)*, pages 161–168. ACM, 1998.

[12] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2016.

[13] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 711–722, 2018.

[14] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *European Conference on Parallel Processing (Euro-Par)*, pages 659–670, 2013.

[15] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 501–510, 2008.

[16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[17] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–12, 2015.

[18] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, pages 14:1–14:18, 2016.

[19] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.

[20] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010.

[21] Guy E Blelloch and Yan Gu. Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 105–119. SIAM, 2020.

[22] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016.

[23] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 235–246, 2018.

[24] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 103–115, 2020.

[25] Guy E. Blelloch, Yan Gu, and Yihan Sun. Efficient construction of probabilistic tree embeddings. In *ICALP*, pages 26:1–26:14, 2017.

[26] Guy E Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest-paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[27] Guy E Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–213, 2012.

[28] Guy E Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–213, 2012.

[29] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

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

[31] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[32] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[33] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[34] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658, 2016.

[35] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha V. Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[36] S Chakraborty, K Sadakane, and S.R. Satti. Optimal in-place algorithms for basic graph problems. *Combinatorial Algorithms*, 2020.

[37] Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the âĂIJrestoreâĂİ model. *ACM Trans. Algorithms*, 14(2), 2018.

[38] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[39] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[40] Edith Cohen and Haim Kaplan. Efficient estimation algorithms for neighborhood variance and other moments. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 157–166, 2004.

[41] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m \log^{1/2} n$ time. In *ACM Symposium on Theory of Computing (STOC)*, pages 343–352, 2014.

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

[43] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*, 2017.

[44] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 393–404, 2018.

[45] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for NVRAMs. *PVLDB*, 13(9), 2020.

[46] Nan Du, Le Song, Manuel Gomez-Rodriguez, and Hongyuan Zha. Scalable influence estimation in continuous-time diffusion networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3147–3155, 2013.

[47] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *ACM Symposium on Theory of Computing (STOC)*, pages 448–455. ACM, 2003.

[48] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[49] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[50] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *PVLDB*, 13(8), 2020.

[51] Yan Gu, Omar Obeya, and Julian Shun. Parallel in-place algorithms: Theory and practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021. To appear.

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

[53] Yan Gu, Yihan Sun, and Guy E. Blelloch. Algorithmic building blocks for asymmetric memories. In *European Symposium on Algorithms (ESA)*, 2018.

[54] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

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

[56] Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

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

[58] Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing*, 25(3):189–205, 2012.

[59] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[60] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. 2014.

[61] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, pages 723–735, 2002.

[62] Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *SPAA*, 2013.

[63] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[64] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multi-threaded asynchronous graph traversal for in-memory and semi-external memory. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[65] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *ACM Symposium on Theory of Computing (STOC)*, pages 255–264, 2008.

[66] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020.

[67] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Principles and Practice of Parallel Programming*, 2013.

[68] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *SPAA*, 2013.

[69] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *SPAA*, 2014.

[70] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, 2015.

[71] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *ICDE*, 2015.

[72] Nodari Sitchinava. Computational geometry in the parallel external memory model. *SIGSPATIAL Special*, 4(2), 2012.

[73] Thomas H Spencer. Time-work tradeoffs for parallel algorithms. *Journal of the ACM (JACM)*, 44(5):742–778, 1997.

[74] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. GraphMP: An efficient semi-external-memory big graph processing system on a single machine. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 276–283, 2017.

[75] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[76] Jeffrey D Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[77] Alexander v Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *International Workshop on Data Management on New Hardware*, page 12. ACM, 2019.

[78] Stratis D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[79] Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Comput. Surveys*, 33(2), 2001.

[80] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, 2015. USENIX Association.

[81] Da Zheng, Disa Mhembere, Vince Lyzinski, Joshua T. Vogelstein, Carey E. Priebe, and Randal Burns. Semi-external memory sparse matrix multiplication for billion-node graphs. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1470–1483, May 2017.