Implicit Decomposition for Write-Efficient Connectivity Algorithms

Naama Ben-David* Guy E. Blelloch* Jeremy T. Fineman†
Phillip B. Gibbons* Yan Gu* Charles McGuffey* Julian Shun‡
*Carnegie Mellon University †Georgetown University ‡MIT CSAIL

***Abstract*—The future of main memory appears to lie in the direction of new technologies that provide strong capacity-to-performance ratios, but have write operations that are much more expensive than reads in terms of latency, bandwidth, and energy. Motivated by this trend, we propose sequential and parallel algorithms to solve graph connectivity problems using significantly fewer writes than conventional algorithms. Our primary algorithmic tool is the construction of an $o(n)$-sized *implicit decomposition* of a bounded-degree graph G on n nodes, which combined with read-only access to G enables fast answers to connectivity and biconnectivity queries on G. The construction breaks the linear-write "barrier", resulting in costs that are asymptotically lower than conventional algorithms while adding only a modest cost to querying time. For general non-sparse graphs on m edges, we also provide the first parallel algorithms for connectivity and biconnectivity that require $o(m)$ writes and $O(m)$ operations. These algorithms provide insight into how applications can efficiently process computations on large graphs in systems with read-write asymmetry.**

I. Introduction

Recent trends in computer memory technologies suggest wide adoption of memory systems in which reading from memory is significantly cheaper than writing to it, especially with regards to energy. The reason for this asymmetry, roughly speaking, is that writing to memory requires a change to the state of the material, while reading only requires detecting the current state.[1] This trend poses the interesting question of how to design algorithms that are more efficient in the number of writes than in the number of reads. To this end recent works have studied models that account for asymmetric read-write costs and have analyzed a variety of algorithms in these models [5], [6], [9]–[11], [13], [18], [19], [24], [30], [36], [37].

Some of this work has shown an inherent tradeoff between reads and writes. For example, Blelloch et al. [10] show that for computations on certain DAGs, any reduction in writes requires an increase in reads. For FFTs and sorting networks, for example, the increase in reads is exponential with respect to the decrease in writes. Intuitively, the tradeoff is because reducing writes restricts the ability to save partial results needed in multiple places, and hence requires repeating certain computations. This is reminiscent of well-studied time-space tradeoffs [23]—but is not the same, because time-space models still allow an *arbitrary* number of writes to the limited space, with each write costing the same as a read.

The full version of this paper is available at arXiv:1710.02637 [7].

[1]See Appendix A in [7] for some technical details of the new memories.

In this paper we are interested in graph connectivity problems, and in particular we are interested in whether it is possible to build an "oracle" using a sublinear number of writes that supports fast queries, along with any read-write tradeoffs this entails. We consider undirected connectivity (connected components and spanning forests) and biconnectivity (biconnected components, articulation points, and related 1-edge-connectivity) problems. We do not consider the cost of initially storing the graph in memory, but note that there are many scenarios in which the graph is either represented implicitly, e.g., the Swendsen-Wang algorithm [34], or for which the graph is sampled and used multiple times, e.g., edges selected based on different Boolean hash functions or based on properties (timestamp, weight, relationship, etc.) associated with the edge.

Our results show that if a graph with n vertices and m edges is sufficiently dense, a sublinear number of writes ($o(m)$) can be achieved without asymptotically increasing the number of reads (no tradeoff is required). For bounded-degree graphs, on the other hand, our algorithm achieving a sublinear number of writes ($o(n)$) involves a linear tradeoff between reads and writes. The main technical contribution is a new *implicit* decomposition of a graph that allows writing out information for only a suitably small *sample* of the vertices. We use two models to account for the read-write asymmetry: (i) the *Asymmetric RAM model* [10], in which writes to the asymmetric memory cost $\omega \gg 1$ and all other operations are unit cost; and (ii) its parallel variant, the *Asymmetric NP* model [6]. Both models have a small symmetric memory (a cache) that can be used to help minimize the number of writes to the large asymmetric memory.

Table I summarizes our main results for these models, showing asymptotic improvements in construction costs over all prior work (sequential or parallel) for these well-studied connectivity problems.

Algorithms with $o(m)$ writes for non-sparse graphs. The first contribution of this paper is a group of algorithms that achieve $O(m/\omega+n)$ writes, $O(m)$ other operations, and hence $O(m+\omega n)$ work. While standard sequential BFS- or DFS-based graph connectivity algorithms require only $O(n)$ writes, and hence already achieve this bound, the parallel setting is more challenging. Existing linear-work parallel connectivity algorithms perform $\Theta(m)$ writes [16], [20]–[22], [31]–[33], and hence are $\Theta(\omega m)$ work in the asymmetric memory setting. We show how the algorithm of Shun et al. [33] can be adapted to use only $O(m/\omega+n)$ writes (and $O(m)$ other operations), by

	Connectivity			Biconnectivity			Best choice
	Seq. time	Parallel work	Query time	Seq. time	Parallel work	Query time	when
Prior work	$O(m+\omega n)$	$O(\omega m)^\dagger$	$O(1)$	$O(\omega m)$	$O(\omega m)^\dagger$	$O(1)$	–
Ours [§IV-B,§V-B]	$O(m+\omega n)^\dagger$	$O(m+\omega n)^\dagger$	$O(1)$	$O(m+\omega n)^\dagger$	$O(m+\omega n)^\dagger$	$O(1)$	$m \in \Omega(\sqrt{\omega}n)$
Ours [§IV-C,§V-C]	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega})^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$O(\omega)^\dagger$	$m \in o(\sqrt{\omega}n)$

TABLE I: Summary of main results for constructing connectivity oracles (n nodes, m edges, †=expected), where $\omega \gg 1$ is the cost of writes to the asymmetric memory. Compared to prior work, asymmetric memory writes are reduced by up to a factor of ω, yielding improvements in both sequential time and parallel work. All parallel algorithms have depth polynomial in $\omega \log n$. For all algorithms the small symmetric memory is only $O(\omega \log n)$ words.

avoiding repeated graph contractions and using a recent write-efficient BFS [6], yielding the first $O(m+\omega n)$ expected work, low-depth parallel algorithm for connectivity in the asymmetric setting. (By ***low-depth*** we mean depth polynomial in $\omega \log n$.)

For biconnectivity, the *standard* output is an array of size m indicating to which biconnected component each edge belongs [17], [25]. Producing this output requires at least m writes, and as a result, the sequential time (and parallel work) ends up being $\Theta(\omega m)$ in the asymmetric memory setting. We present an equally effective representation of the output, which we call the *BC labeling*, which has size only $O(n)$. This leads to a sequential biconnectivity algorithm that constructs the oracle in only $O(m+\omega n)$ time in the asymmetric setting. Moreover, we show how to leverage our new parallel connectivity algorithm to compute the BC labeling in $O(m/\omega + n)$ writes, yielding the first $O(m+\omega n)$ work parallel algorithm for biconnectivity in the asymmetric memory setting. We show:

Theorem I.1. *Graph connectivity and biconnectivity oracles can be constructed in parallel with $O(m+\omega n)$ expected work and $O(\omega^2 \log^2 n)$ depth whp[2] on the Asymmetric NP model, and each query can be answered in $O(1)$ work. Sequentially, the construction takes $O(m+\omega n)$ time on the Asymmetric RAM model, with $O(1)$ query time. The symmetric memory used is $O(\omega \log n)$ words.*

Algorithms with $o(n)$ writes for sparse graphs. For sparse graphs, the work of our connectivity and biconnectivity algorithms is dominated by the $\Omega(n)$ writes they perform. This led us to explore the following fundamental question: ***Is it possible to construct, using $o(n)$ writes to the asymmetric memory, an oracle for graph connectivity (or biconnectivity) that can answer queries in time independent of n?*** Given that the standard output for these problems (even with BC labeling) is $\Theta(n)$ size even for bounded-degree graphs, one might conjecture that $\Omega(n)$ writes are required. Our main contribution is a (perhaps surprising) affirmative answer to the above question for both the connectivity and biconnectivity problems.

The key technical contribution behind our breaking of the $\Omega(n)$-write "barrier" is the definition and use of an ***implicit k-decomposition*** of a graph. Informally, a *k-decomposition* of a graph G is comprised of a subset S of the vertices, called *centers*, and a mapping $\rho(\cdot)$ that partitions the vertices among the centers, such that (i) $|S| = O(n/k)$, (ii) the number of vertices in each partition is at most k, and (iii) for each center, the induced subgraph on its vertices is connected. However, explicitly storing the center associated with each vertex would require $\Omega(n)$ writes. Instead, an *implicit k-decomposition* defines the mapping implicitly in terms of a procedure that is given only G and S (and a 1-bit labeling on S).

With the new concept of implicit k-decomposition, we present three algorithmic subroutines which together construct connectivity and biconnectivity oracles with $O(m/\sqrt{\omega})$ writes, which is $o(n)$ when $m \in o(\sqrt{\omega}n)$. For clarity of presentation, we begin by assuming the input graph has bounded degree. Section VI discusses how to relax this constraint.

We first present an algorithm to compute an implicit k-decomposition that can be constructed in only $O(n/k)$ writes, $O(kn)$ reads, and low depth, and can compute $\rho(v)$ in only $O(k)$ expected reads and no asymmetric memory writes. The intuition behind our construction is first to pick a random subset of the vertices and then map each unpicked vertex to the closest center by performing a BFS on the graph G. Unfortunately, this does not satisfy the constraint on partition size, so a more sophisticated approach is needed. The unique challenge that arises again and again in the asymmetric context is that the sublinear limitation on the number of writes rules out the approaches used by prior work.

We then show how the implicit k-decomposition can be used to solve graph connectivity and biconnectivity. We define the concept of a *clusters graph*, which contains vertices each representing a cluster and edges between clusters. The key idea is that after precomputing on the clusters graph and storing a constant amount of information about connectivity and biconnectivity on each vertex (corresponding to a cluster in the original graph), a connectivity or biconnectivity query can be answered from only the local structure and preprocessed information on a constant number of clusters. This is straightforward for connectivity queries because we need only compare the labels of the clusters that contain the respective query points. However, this becomes much more challenging in graph biconnectivity because the correspondence between the clusters and biconnected components is non-trivial: a cluster may contain the vertices in many biconnected components while the vertices in a certain biconnected components can belong to different clusters. Therefore, biconnectivity queries require considerable subtleties in the design, to store the appropriate information on the clusters graph to enable each query to access only a constant number of clusters. More specifically, we define the concept of the *local graph* of each cluster (it maintains the relationship of biconnectivity of the

[2]Throughout the paper we use *whp* to mean with probability $1 - n^{-c}$ for any constant c that shows up linearly in the cost bound (e.g. $O(c\omega^2 \log^2 n)$ in the bound given).

cluster and its neighboring clusters and can be computed with cost proportional to the size of the cluster), such that the biconnectivity queries discussed in Section V can be answered from a constant number of local graphs and the information stored in the clusters graph.

Our sequential algorithms have significant algorithmic merits on their own, but we also show that all the algorithms can be made to run in parallel with low depth. We show:

Theorem I.2. *Graph connectivity and biconnectivity oracles can be constructed in $O(m/\sqrt{\omega})$ expected writes and $O(m\sqrt{\omega})$ expected time (parallel work) on the Asymmetric RAM model (Asymmetric NP model, respectively). The depth on the Asymmetric NP is $O(\omega^{3/2}\log^3 n)$ whp. Each connectivity query can be answered in $O(\omega^{1/2})$ expected time (work) ($O(\omega^{1/2}\log n)$ whp) and each biconnectivity query can be answered in $O(\omega)$ expected time (work) ($O(\omega \log n)$ whp). The symmetric memory used is $O(\omega \log n)$ words.*

II. Preliminaries and Related Work

Let $G = (V, E)$ be an undirected, unweighted graph with $n = |V|$ vertices and $m = |E|$ edges. G can contain self-loops and parallel (duplicate) edges, and is not necessarily connected. We assume a global ordering of the vertices to break ties when necessary. If the degree of every vertex is bounded by a constant, we say the graph is ***bounded-degree***. We use standard definitions of *spanning tree*, *spanning forest*, *connected component*, *biconnected component*, *articulation points*, *bridge*, and *k-edge-connectivity* on a graph, and *lowest-common-ancestor* (LCA) query on a tree (which can be found in [17] and are summarized in Appendix B in [7]). Let $[n] = \{1, 2, \cdots, n\}$ where n is a positive integer.

Computation models. Sequential algorithms are analyzed using the ***Asymmetric RAM*** model [10], comprised of an infinitely large asymmetric memory and a small symmetric memory. The cost of writing to the large memory is ω, and all other ***operations*** (instructions) have unit cost. This models practical settings in which there is a small amount of standard symmetric memory (e.g., a cache) in addition to the asymmetric memory.

For parallel algorithms, we use the ***Asymmetric Nested-Parallel (NP)*** model [6], which is designed to characterize both parallelism and memory read-write asymmetry. In the model, a computation is represented as a (dynamically unfolding) directed acyclic graph (DAG) of tasks that begins and ends with a single task called the root. A task consists of a sequence of instructions that must be executed in order. Tasks can also call the Fork instruction, which creates child tasks that can be run in parallel with each other. The memory in the Asymmetric NP Model consists of (i) an infinitely large *asymmetric* memory accessible to all tasks and (ii) a small task-specific *symmetric* memory accessible only to a task and its children. The cost of writing to large memory is ω, and all other operations have unit cost. The ***work*** W of a computation is the sum of the costs of the operations in its DAG and the ***depth*** D is the cost of the DAG's most expensive path. Under mild assumptions, Ben-David et al. [6] show that a work-stealing scheduler can execute an algorithm whose Asymmetric NP complexity is W work and D depth in $O(W/P + \omega D)$ expected time on P processors.

In both models, the number of ***writes*** refers only to the writes to the asymmetric memory, ignoring writes to symmetric memory. In general we use ***operations*** to include all instructions other than a write to asymmetric memory. All reads and writes are to words of size $\Theta(\log n)$ for an input size of n. The size of the symmetric memory is measured in words. For queries we do not include the cost of writing the result to asymmetric memory because the query might be part of a larger query done in symmetric memory.

Related Work. Read-write asymmetries have been studied in the context of NAND Flash chips [5], [18], [19], [30], focusing on how to balance the writes across the chip to avoid uneven wear-out of locations. Targeting instead the new memory technologies, read-write asymmetries have been an active area of research in the systems/database/architecture communities (e.g., [2], [4], [13]–[15], [26], [29], [36]–[42]). In the algorithms community, Blelloch et al. [9] define several sequential and parallel computation models that take asymmetric read-write costs into account, and design efficient sorting algorithms under these models. Their follow-up paper [10] presents sequential algorithms for various problems that perform better than their classic counterparts under asymmetric read-write costs, as well as several lower bounds. Carson et al. [11] present write-efficient sequential algorithms for a similar model, as well as write-efficient parallel algorithms (and lower bounds) on a distributed memory model with asymmetric read-write costs, focusing on linear algebra problems and direct N-body methods. Ben-David et al. [6] propose the Asymmetric NP model and present write-efficient, work-efficient, low depth (span) parallel algorithms for reduce, list contraction, tree contraction, breadth-first search, ordered filter, and planar convex hull, as well as a write-efficient, low-depth minimum spanning tree algorithm that is nearly work-efficient. Jacob and Sitchinava [24] prove lower bounds for an asymmetric external memory model. In each of these models, there is a small amount of symmetric memory that can be used to help minimize the number of writes to the large asymmetric memory.

Although graph decompositions with various properties have been shown to be quite useful in a large variety of applications (e.g., [3], [27], [28]), to our knowledge none of the prior algorithms provide the necessary conditions for processing graphs with a sublinear number of writes to answer connectivity/biconnectivity queries (targeting instead other decomposition properties that are unnecessary in our setting, such as having few edges between clusters). For example, Miller et al.'s [28] parallel low-diameter decomposition algorithm requires at least $\Omega(n)$ writes (even if a write-efficient BFS [6] is used), and provides no guarantees on the partition sizes. Similarly, algorithms for size-balanced graph partitioning (e.g., [1]) require $\Omega(n)$ writes. Our implicit k-decomposition construction is reminiscient of sublinear time algorithms for

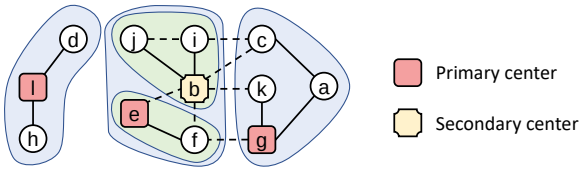

Fig. 1: An example implicit k-decomposition for $k = 4$ consisting of clusters $\{d, h, l\}, \{i, j, b\}, \{e, f\}$, and $\{a, c, g, k\}$. In the graph, $j's$ primary center is e (i.e., $\rho_0(j) = e$) and its secondary center is b (i.e., $\rho(j) = b$). Note b is on the shortest path from j to e. Also note that c is closer to the secondary center b than to g, but picks g as its actual center, because b is not on the shortest path to its primary center. In breaking ties we assume lexicographically smaller letters have higher priorities. The solid lines are on shortest paths from a vertex to its secondary center.

estimating the number of connected components [8], [12] in its use of BFS from a sample of the vertices. However, their BFS is used for a completely different purpose (approximate counting of $1/n_u$, the inverse of the size of the connected component containing a sampled node u), does not provide a partitioning of the nodes into clusters (two BFS's from different nodes can overlap), and cannot be used for connectivity or biconnectivity queries (two BFS's from the same connected component may be truncated before intersecting).

III. Implicit Decomposition

In this paper we introduce the concept of an implicit decomposition. The idea is to partition an undirected graph into connected clusters such that all we need to store to represent the cluster is one representative, which we call the center of the cluster, and some small amount of information on that center (1 bit in our case). The goal is to quickly answer queries on the cluster. The queries we consider are: given a vertex find its center, and given a center find all its vertices. To reduce the amount of symmetric-memory needed, we need all clusters to be roughly the same size. We start with some definitions, which consider an undirected graph G.

For graph $G = (V, E)$ we refer to the subgraph induced by a subset of vertices as a ***cluster***. A ***decomposition*** of a connected graph $G = (V, E)$ is a vertex subset $S \subset V$, called ***centers***, and a function $\rho(v) : V \to S$, such that the ***cluster*** $C(s) = \{v \in V \mid \rho(v) = s\}$ for each center $s \in S$ is connected. A decomposition is a ***k-decomposition*** if the size of each cluster is upper bounded by k, and $|S| = O(n/k)$. We are often interested in the graph induced by the decomposition, and in particular:

Definition 1. *Given the decomposition (S, ρ) of a graph $G = (V, E)$, the* **clusters graph** *is the multigraph $G' = (S, \langle\ \{\rho(u), \rho(v)\} : \{u, v\} \in E, \rho(u) \neq \rho(v)\ \rangle\ )$.*

Definition 2. *An* **implicit decomposition** *of a connected graph $G = (V, E)$ is a decomposition (S, ρ, ℓ) such that $\rho(\cdot)$ is defined implicitly in terms of an algorithm given only G, S, and a (short) labeling $\ell(s)$ on $s \in S$.*

In this paper, we use implicit k-decompositions. Our goal is to construct and query the decomposition quickly, while using short labels. Our main result is the following.

Theorem III.1. *An implicit k-decomposition (S, ρ, ℓ) can be constructed on an undirected bounded-degree graph $G = (V, E)$ with $|V| = n$ such that:*

- *the construction takes $O(kn)$ operations and $O(n/k)$ writes, both in expectation;*
- *$\rho(v)$ query: finding $\rho(v)$ for any given $v \in V$ takes $O(k)$ operations in expectation and $O(k \log n)$ whp, and no writes;*
- *$C(s)$ query: finding $C(s)$ for any given $s \in S$ takes $O(k^2)$ operations in expectation, and $O(k^2 \log n)$ whp and no writes;*
- *the labels $\ell(s), s \in S$ are 1-bit each; and,*
- *construction and queries take $O(k \log n)$ symmetric memory whp.*

Note that this theorem indicates a tradeoff between reads (operations) and writes for the construction controlled by k.

At a high level, the construction algorithm works by identifying a subset of centers such that every vertex can quickly find its nearest center without having to keep a pointer to it (which would require too many writes). It first selects a random subset of the vertices where each vertex is selected with probability $1/k$. We call these the ***primary centers*** and denote them as S_0. All other vertices are then assigned to the nearest such center. Unfortunately, a cluster defined in this way can be significantly larger than k (super-polynomial in k). To handle this, the algorithm identifies an additional $O(n/k)$ ***secondary centers***, S_1. Every vertex v is associated with a primary center $\rho_0(v) \in S_0$, and an actual center $\rho(v) \in S = S_0 \cup S_1$. The only values the algorithm stores are the set S and the bits $\ell(s), s \in S$ indicating whether it is primary or secondary. An example is given in Figure 1.

In our construction it is important to break ties among equal-length paths in a consistent way, such that subpaths of a shortest path are themselves a unique shortest path. For this purpose we assume the vertices have a total ordering (and comparing two vertices takes constant time). Among two equal hop-length paths from a vertex u, consider the first vertex where the paths diverge. We say that the path with the higher priority vertex at that position is shorter. Let $SP(u, v)$ be the shortest path between u and v under this definition for breaking ties, and $L(SP(u, v))$ be its length such that comparing $L(SP(u, v))$ and $L(SP(u, w))$ breaks ties as defined. By our definition all subpaths of a shortest path are also unique shortest paths for a fixed vertex ordering. Based on these definitions we specify $\rho_0(v)$ and ρ as follows:

$$\rho_0(v) = \operatorname*{argmin}_{u \in S_0} L(v, u)$$

$$\rho(v) = \operatorname*{argmin}_{u \in S \wedge u \in SP(v, \rho_0(v))} L(v, u)$$

The definitions indicate that a vertex's center is the first center encountered on the shortest path to the nearest primary center. This could either be a primary or secondary center (see Figure 1). $\rho(v)$ is defined in this manner to prevent vertices

from being reassigned to secondary centers created in other primary clusters, which could result in oversized clusters.

We now describe how to find $\rho(v)$ for any vertex $v \in V$. First, we find v's closest primary center by running a BFS from v until we hit a vertex in S_0. The BFS orders the vertices by $L(SP(v,\cdot))$. To find $\rho(v)$ we first search for the primary center of v ($\rho_0(v)$) and then identify the first center on the path from v to $\rho_0(v)$, possibly $\rho_0(v)$ itself.

Lemma III.2. *$\rho(v)$ can be found in $O(k)$ operations in expectation, and $O(k \log n)$ operations whp, and using $O(k \log n)$ symmetric memory whp.*

Proof. Note that the search order from a vertex is deterministic and independent of the sampling used to select S_0. Therefore, the expected number of vertices visited before hitting a vertex in S_0 is k. By tail bounds, the probability of visiting $O(ck \log n)$ vertices before hitting one in S_0 is at most $1/n^c$. The search is a BFS, so it takes time linear in the number of vertices visited. Because the vertices are of bounded degree, placing them in priority order in the queue is easy. Once the primary center is found, a search back on the path gives the actual center. We assume that symmetric memory is used for the search so that no writes to the asymmetric memory are required. The memory used is proportional to the search size, which is proportional to the number of operations; $O(k)$ in expectation and $O(k \log n)$ *whp*. □

The space requirement for the symmetric memory is $O(k \log n)$, which we believe is realistic as we set $k = \sqrt{\omega}$ when using this decomposition later in this paper.

We use the following lemma to help find $C(s)$ for a center s.

Lemma III.3. *The union of the shortest paths* $\mathrm{SP}(v, \rho(v))$ *for $v \in V$ define a rooted spanning tree on each cluster, with the center as the root (when path edges are viewed as directed towards $\rho(v)$).*

Proof. We first show that this is true for the clusters defined by the primary centers S (i.e., $\rho_0(v)$). We use the notation $SP(v,u) + SP(u,w)$ to indicate joining the two shortest paths at u. Consider a vertex v with $\rho_0(v) = s$, and consider all of the vertices P on the shortest path from v to s. The claim is that for each $u \in P, \rho(u) = s$ and $SP(u,s)$ is a subpath of P. This implies a rooted tree. To see that $\rho(u) = s$ note that the shortest path from u to a primary vertex t has length $L(SP(u,t))$. We can write the length of the shortest path from v to t as $L(SP(v,t)) \leq L(SP(v,u) + SP(u,t))$ and the length of the shortest path from v to s as $L(SP(v,s)) = L(SP(v,u) + SP(u,s))$. Because $\rho_0(v) = s$, we know that $L(SP(v,s)) < L(SP(v,t))\ \forall t \neq s$. Through substitution and subtraction, we see that $L(SP(u,s)) < L(SP(u,t))\ \forall t \neq s$. This means that $\rho_0(u) = s$. We know that $SP(u,s)$ cannot contain the edge b that v takes to reach u in $SP(v,s)$ because $u \in SP(v,s)$. Since the search from u excluding b will have the same priorities as the search from v when it reaches u, $SP(u,s)$ is a subpath of P.

Algorithm 1: Constructing Implicit k-Decomposition

Input: Connected bounded-degree graph $G = (V, E)$, parameter k
Output: A set of cluster centers S_0 and S_1 ($S = S_0 \bigcup S_1$)

```
Sample each vertex with probability 1/k, and place in S_0
S_1 = ∅
foreach vertex v ∈ S_0 do
    SecondaryCenters(v, G, S_0)
return S_0 and S_1
function SecondaryCenters(v, G, S)
    Search from v for the first k vertices that have v as their
      center. This defines a tree.
    If the search exhausts all vertices with center v, return.
    Otherwise identify a vertex u that partitions the tree such
      that its subtree and the rest of the tree are each at least a
      constant fraction of k.
    Add u to S_1.
    SecondaryCenters(v, G, S ∪ u)
    SecondaryCenters(u, G, S ∪ u)
```

Now consider the clusters defined by $\rho(v)$. The secondary centers associated with a primary center s partition the tree for s into subtrees. Each subtree begins (relative to the root) at a center and ends when encountering another center (this other center is not included). Each vertex in the tree for s will be assigned the correct partition by $\rho(v)$ because each will be assigned to the first secondary center on the way to the primary center. □

The set of solid edges in Figure 1 is an example of the spanning forest. This gives the following.

Corollary III.4. *For any vertex v,* $\mathrm{SP}(v, \rho(v)) \subseteq C(\rho(v))$.

Lemma III.5. *For any vertex $s \in S$, its cluster $C(s)$ can be found in $O(k|C(s)|)$ operations in expectation and $O(k|C(s)| \log n)$ operations whp, and using $O(|C(v)| + k \log n)$ symmetric memory whp.*

Proof. For any center $s \in S$, identifying all the vertices in its cluster $C(s)$ can be implemented as a BFS starting at s. For each vertex $v \in V$ that the BFS visits, the algorithm checks if $\rho(v) = s$. If so, we add v to $C(s)$ and put its unvisited neighbors in the BFS queue, and otherwise we do neither. By Corollary III.4, any vertex v for which $\rho(v) = s$ must have a path to s only through other vertices whose center is v. Therefore the algorithm will visit all vertices in $C(s)$. Furthermore, because the graph has bounded degree it will only visit $O(C(s))$ vertices not in $C(s)$. Each visit to a vertex u requires finding $\rho(v)$. Our bound on the number of operations therefore follows from Lemma III.2. We use $O(|C(v)|)$ symmetric memory for storing the queue and $C(v)$, and $O(k \log n)$ memory *whp* for calculating $\rho(v)$. □

We now show how to select the secondary centers such that the size of each cluster is at most k. Algorithm 1 describes the process. By Lemma III.3, before the secondary centers are added, each primary vertex in $s \in S_0$ defines a rooted tree of paths from the vertices in its cluster to s. The function SecondaryCenters then recursively cuts up this tree into subtrees rooted at each u that is identified.

Lemma III.6. *Algorithm 1 runs in $O(nk)$ operations and $O(n/k)$ writes (both in expectation), and uses $O(k \log n)$ symmetric memory whp on the Asymmetric RAM Model. It generates a implicit k-decomposition S of G with labels distinguishing S_0 from S_1.*

Proof. The algorithm creates clusters of size at most k by construction (it partitions any cluster bigger than k using the added vertices u). Each call to SECONDARYCENTERS (without recursive calls) will use $O(k^2)$ operations in expectation because we visit k vertices and each one has to search back to v to determine if v is its center. Each call also uses $O(k \log n)$ space for the search *whp* because we need to store the k elements found so far and each $\rho(v)$ uses $O(k \log n)$ space for the search *whp*. Before making the recursive calls, we write out u to S_1, requiring one write per call to SECONDARYCENTERS. The symmetric memory can be reused by the recursive calls.

We are left with showing there are at most $O(n/k)$ calls to SECONDARYCENTERS. There are n/k primary centers in expectation. If there are too many (beyond some constant factor above the expectation), we can try again. Because the graph has bounded degree, we can find a vertex that partitions the tree such that its subtree and the rest of the tree are both at most a constant fraction of k. We can now charge all internal nodes of the recursion against the leaves. There are at most $O(n/k)$ leaves because each defines a cluster of size $\Theta(k)$. Therefore there are $O(n/k)$ calls to SECONDARYCENTERS, giving the stated overall bounds. □

Parallelizing the decomposition. To parallelize the decomposition in Algorithm 1, we make one small change; in addition to adding u to the set of secondary centers at each recursive call to SECONDARYCENTERS, we add all children of v (thus separating v into its own cluster). This guarantees that the height of the tree decreases by at least one on each recursive call, and only increases the number of writes by a constant factor. This gives the following lemma.

Lemma III.7. *On the Asymmetric NP model, Algorithm 1 runs in depth $O((k \log n)(k^2 \log n + \omega))$ whp.*

Proof. Certainly selecting the set S_0 can be done in parallel. Furthermore the calls to SECONDARYCENTERS on line 4 can be made recursively in parallel. The depth will be proportional to the depth to each call to SECONDARYCENTERS (not including recursive calls) multiplied by the depth of the recursion. To bound the depth, in the parallel version we also mark the children of the root as secondary centers, which does not increase the number of secondary centers asymptotically (due to the bounded-degree assumption). In this way the height of the tree decreases by one on each recursive call. The depth of the recursion is at most the depth of the tree associated with the primary center $\rho_0(v)$. This is bounded by $O(k \log n)$ *whp* because by Lemma III.2 every vertex finds its primary center within $O(k \log n)$ steps *whp*. The depth of SECONDARYCENTERS (without recursion) is just the number of operations ($O(k^2 \log n)$ *whp*) plus the depth of the one write of u (which costs ω). This gives the bound. □

Extension to unconnected graphs. Note that once a connected component contains at least one primary center, the definition and Theorem III.1 hold. However, it is possible that in a small component, the search of $\rho(\cdot)$ exhausts all connected vertices without finding any primary centers (vertices in the initial sample, S_0). In this case, we check whether the size of the cluster is at least k, and if so, we mark as a primary center the vertex that is the smallest according to the total order on vertices. This marks at most n/k primary centers and the rest of the algorithm remains unchanged. This step is added after line 1 in Algorithm 1, and requires $O(nk)$ work and operations, $O(n/k)$ writes, and $O(k)$ depth. The cost bound therefore is not changed. If the component is smaller than k, we use the smallest vertex in the component as a center implicitly, but never write it out. The $\rho(\cdot)$ function can easily return this in $O(k)$ operations.

IV. GRAPH CONNECTIVITY AND SPANNING FOREST

This section describes parallel write-efficient algorithms for graph connectivity and spanning forest; that is, identifying which vertices belong to each connected component and producing a spanning forest of the graph. These tasks can be easily accomplished sequentially by performing a breadth-first or depth-first search in the graph with $O(m)$ operations and $O(n)$ writes. While there are several work-efficient parallel algorithms for the problem [16], [20]–[22], [31]–[33], all of them use $\Omega(n + m)$ writes. This section has two main contributions: (1) Section IV-B provides a parallel algorithm using $O(n+m/\omega)$ writes in expectation, $O(n\omega+m)$ expected work, and $O(\omega^2 \log^2 n)$ depth with high probability; (2) Section IV-C gives an algorithm for constructing a connectivity oracle on constant-degree graphs in $O(n/\sqrt{\omega})$ expected writes and $O(n\sqrt{\omega})$ expected total operations. Our oracle-construction algorithm is parallel, having depth $O(\omega^{3/2} \log^3 n)$ *whp*, but it also represents a contribution as a sequential algorithm.

Our parallel algorithm (Section IV-B) can be viewed as a write-efficient version of the parallel algorithm due to Shun et al. [33]. This algorithm uses a low-diameter decomposition algorithm of Miller et al. [28] as a subroutine, which we review and adapt next in Section IV-A and Appendix C in [7].

A. Low-diameter Decomposition

Here we review the low-diameter decomposition of Miller et al. [28]. The so-called "(β, d)-decomposition" is terminology lifted from their paper, and it should not be confused with our implicit k-decompositions. The details of the decomposition subroutine are important only to extract a bound on the number of writes, and are left to Appendix C in [7].

A (β, d)-***decomposition*** of an undirected graph $G = (V, E)$, where $0 < \beta < 1$ and $1 \leq d \leq n$, is defined as a partition of V into subsets $V_1, \ldots, V_\ell$ such that (1) the shortest path between any two vertices in each V_i using only vertices in V_i has length at most d, and (2) the number of edges $(u, v) \in E$ crossing the partition, i.e., such that $u \in V_i$, $v \in V_j$, and $i \neq j$, is at most βm. Miller et al. [28] provide a parallel algorithm for generating a $(\beta, O(\log n/\beta))$-decomposition using $O(m)$

operations, reads, and writes. As described, however, their algorithm performs $\Theta(m)$ writes. The key subroutine of the algorithm, however, is just breadth-first searches (BFS's). Replacing these BFS's with write-efficient BFS's [6] yields the following theorem:

Theorem IV.1. *A $(\beta, O(\log n/\beta))$-decomposition can be generated in $O(n)$ expected writes, $O(m + \omega n)$ expected work, and $O(\omega \log^2 n/\beta)$ depth whp on the Asymmetric NP model.*

B. Connectivity and Spanning Forest

The parallel connectivity algorithm of [33] applies the low-diameter decomposition recursively with β set to a constant less than 1. Each level of recursion contracts a subset of vertices into a single supervertex for the next level. The algorithm terminates when each connected component is reduced to a single supervertex. The stumbling block for write efficiency is this contraction step, which performs writes proportional to the number of remaining edges.

Instead, our write-efficient algorithm applies the low-diameter decomposition just once, but with a much smaller β, as follows:

1) Perform the low-diameter decomposition with parameter $\beta = 1/\omega$.
2) Find a spanning tree on each V_i (in parallel) using write-efficient BFS's of [6].
3) Create a contracted graph, where each vertex subset in the decomposition is contracted down to a single vertex. To write down the cross-subset edges in a compacted array, employ the write-efficient filter of [6].
4) Run any parallel linear-work spanning forest algorithm on the contracted graph, e.g., the algorithm from [16] with $O(\omega \log n)$ depth.

Combining the spanning forest edges across subsets (produced in Step 4) with the spanning tree edges (produced in Step 2) gives a spanning forest on the original graph. Adding the bounds for each step together yields the following theorem. Again only $O(1)$ symmetric memory is required per task.

Theorem IV.2. *For any choice of $0 < \beta < 1$, connectivity and spanning forest can be solved in $O(n + \beta m)$ expected writes, $O(\omega n + \beta \omega m + m)$ expected work, and $O(\omega \log^2 n/\beta)$ depth whp on the Asymmetric NP model. For $\beta = 1/\omega + n/m$, these bounds reduce to $O(n + m/\omega)$ expected writes, $O(m + \omega n)$ expected work and $O(\omega \min\{\omega, m/n\} \log^2 n)$ depth whp.*

The proof is given in the full version of this paper [7].

C. Connectivity Oracle in Sublinear Writes

A connectivity oracle supports queries that take as input a vertex and return the label (component ID) of the vertex. This allows one to determine whether two vertices belong in the same component. The algorithm is parameterized by a value k, to be chosen later. We assume throughout that the symmetric memory per task is $\Omega(k \log n)$ words and that the undirected graph has bounded degree.

We begin with an outline of the algorithm. The goal is to produce an oracle that can answer for any vertex which component it belongs to in $O(k)$ work. To build the oracle, we would like to run the connectivity algorithm on the clusters graph produced by an implicit k-decomposition. The result would be that all center vertices in the same component be labeled with the same identifier. Answering a query then amounts to outputting the component ID of the center it maps to, which can be queried in $O(k)$ expected work and $O(k \log n)$ work *whp* according to Lemma III.2.

The main challenge in implementing this strategy is that we cannot afford to write out the edges of the clusters graph (as there could be too many edges). Instead, we treat the implicit k-decomposition as an implicit representation of the clusters graph. Given an implicit representation, our connected components algorithm is the following:

1) Find an implicit k-decomposition of the graph.
2) Run the write-efficient connectivity algorithm from Section IV-B with $\beta = 1/k$, treating the k-decomposition as an implicit representation of the clusters graph, i.e., querying edges as needed.

As used in the connectivity algorithm, our implicit representation need only be able to list the edges in the clusters graph that are adjacent to a center vertex x. To do so, start at x, and explore outwards (e.g., with BFS), keeping all vertices and edges encountered so far in symmetric memory. For each frontier vertex v, query its center (as in Lemma III.5) — if $\rho(v) = x$, then v's unexplored neighbors are added to the next frontier; otherwise (if $\rho(v) \neq x$) the edge $(x, \rho(v))$ is in the clusters graph, so add it to the output list.

Lemma IV.3. *Assuming a symmetric memory of size $\Omega(k \log n)$, the centers neighboring each center in the clusters graph can be listed in no writes, and work, depth, and operations all $O(k^2)$ in expectation or $O(k^2 \log n)$ whp.*

Proof. Listing all the vertices in the cluster takes expected work $O(k^2)$ according to Lemma III.5, or $O(k^2 \log n)$ *whp*. The number of vertices in the cluster is $O(k)$, so they can all fit in symmetric memory. Moreover, because each vertex in the cluster has $O(1)$ neighbors, the total number of explored vertices in neighboring clusters is $O(k)$, all of which can fit in symmetric memory. Each of these vertices is queried with a cost of $O(k)$ operations in expectation and $O(k \log n)$ *whp* given the specified symmetric memory size (Lemma III.2). □

Note that a consequence of the implicit representation is that listing neighbors is more expensive, and thus the number of operations performed by a BFS increases, affecting both the work and the depth. The implicit representation is only necessary while operating on the original clusters graph, i.e., while finding the low-diameter decomposition and spanning trees of each of those vertex subsets; the contracted graph can be built explicitly as before. The best choice of k is $k = \sqrt{\omega}$, giving us the following theorem, whose proof is provided in the full version of this paper [7].

Theorem IV.4. *A connectivity oracle that answers queries in $O(\sqrt{\omega})$ expected work and $O(\sqrt{\omega} \log n)$ work whp can be constructed in $O(n/\sqrt{\omega})$ expected writes, $O(\sqrt{\omega} n)$ expected*

work, and $O(\omega^{3/2}\log^3 n)$ depth whp on the Asymmetric NP model, assuming a symmetric memory of size $\Omega(\sqrt{\omega}\log n)$.

We can also output the spanning forest on the contracted graph in the same bounds, which will be used in the biconnectivity algorithms in Sections V-C and V-D.

V. Graph Biconnectivity

In this section we introduce algorithms related to biconnectivity and 1-edge connectivity queries. We first review the classic approach and its output, which requires $\Theta(m)$ writes. Then we propose a new BC (biconnected-component) labeling output, which has size $O(n)$ and can be constructed in $O(n)$ writes. Queries such as determining bridges, articulation points, and biconnected components can be answered in $O(1)$ operations (and no writes) with the BC labeling. Finally we show how an implicit k-decomposition (as generated by Algorithm 1) can be integrated into the algorithm to further reduce the writes to $O(n/\sqrt{\omega})$.

We begin by presenting sequential algorithms that we believe to be new and interesting. Then in Section V-D we show that these algorithms are parallelizable. All of our algorithms use $O(k\log n)$ symmetric memory.

In this section we assume that the graph is connected. If not, we can run the connectivity algorithm and then run the algorithm on each component. The results for a graph are the union of the results of each of its connected components. Several of the proofs for theorems in this section are given in the full version of this paper [7].

A. The Classic Algorithm

The classic Tarjan-Vishkin parallel algorithm [35] to compute biconnected components and bridges of a connected graph is based on the Euler-tour technique. The algorithm starts by building a spanning tree T rooted at some arbitrary vertex. Each vertex is labeled by *first*(v) and *last*(v), which are the ranks of v's first and last appearance on the Euler tour of T. The low value *low*(v) and the high value *high*(v) of a vertex $v \in V$ are defined as:

$$\textit{low}(v) = \min\{w(u) \mid u \textit{ is in the subtree rooted at } v\}$$
$$\textit{high}(v) = \max\{w(u) \mid u \textit{ is in the subtree rooted at } v\}$$

where

$$w(u) = \min\{\textit{first}(u) \cup \{\textit{first}(u') \mid (u,u') \textit{ is a nontree edge}\}\}^3$$

Namely, *low*(v) and *high*(v) indicate the first and last vertex on the Euler tour that are connected by a nontree edge to the subtree rooted at v. The *low*$(\cdot)$ and *high*$(\cdot)$ values can be computed by a reduce on each vertex followed by a leaffix[4] on the subtrees. The computation of low and high takes $O(\omega\log n)$ depth, $O(m+\omega n)$ work, and $O(n)$ writes on the Asymmetric NP model, by using the tree-contraction algorithm and scheduling theorem from [6]. Then a tree edge is a bridge if and only if the child's *low* and *high* is inclusively within the range of *first* and *last* values of its parent. The cost of this variant of the Tarjan-Vishkin algorithm applied to finding bridges is dominated by the cost of the spanning tree, as given in Theorem IV.2.

[3]If there are multiple edges (u,u') in the graph, none of them are considered here.

[4]Leaffix is similar to prefix but defined on a tree and computed from the leaves to the root.

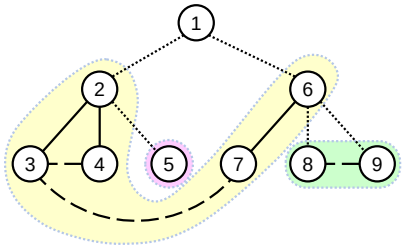

Fig. 2: An example of the BC labeling of a graph. The spanning tree is rooted at vertex 1. The solid and dotted lines indicate tree edges with the dotted lines being the critical edges. Dashed lines are non-tree edges. The vertex labels are $l = [\emptyset, 1, 1, 1, 2, 1, 1, 3, 3]$ (vertex 1 does not have a label), and component heads $r = [1, 2, 6]$. Based on the BC labeling the bridges, articulation points, and biconnected components can be easily retrieved as $\{(2,5)\}$, $\{2,6\}$, and $\{\{1,2,3,4,6,7\},\{2,5\},\{6,8,9\}\}$.

For biconnected components, the standard output is an array $B[\cdot]$ of size m, where the i-th element in B indicates to which biconnected component the i-th edge belongs [17], [25]. Explicitly writing out B is costly in the asymmetric setting, especially when $m \gg n$. We provide an alternative BC labeling as output that requires only $O(n)$ writes.

B. The BC Labeling

Here we describe the ***BC (biconnected-component) labeling***, which effectively represents biconnectivity output in $O(n)$ space. Instead of storing all edges within each biconnected component, the BC labeling stores a component label for each vertex, and a vertex for each component. An example of a BC labeling of a graph is shown in Figure 2. We will later show how to use this representation along with an implicit decomposition to reduce the writes further.

Definition 3. *The* **BC labeling** *of a connected undirected graph with respect to a rooted spanning tree stores a* **vertex label** $l : V\backslash\{\textit{root}\} \to [C]$ *where C is the number of biconnected components in the graph, and a* **component head** $r : [C] \to V$ *of each biconnected component.*

Lemma V.1. *The BC labeling of a connected graph can be computed in $O(m)$ operations and $O(n + m/\omega)$ writes on the Asymmetric RAM. Queries to find bridges, articulation points, or biconnected components can be answered in no writes and $O(1)$ operations given a BC labeling on a rooted spanning tree.*

The algorithm to compute BC labeling. A vertex $v \in V$ (except for the root) is an articulation point if and only if there exists at least one child u in the spanning tree that has *first*$(v) \le$ *low*(u) and *high*$(u) \le$ *last*(v). We call the tree edge between such a pair of vertices a ***critical edge***. The algorithm to compute the BC labeling simply removes all critical edges and

runs graph connectivity on all remaining graph edges. Then the algorithm described in Section IV-B gives a unique component label that we assign as the vertex label. For each component, its head is the vertex that is its parent in the spanning tree (this parent is unique). Each connected component and its head form a biconnected component.

The correctness of the algorithm can be proven by showing the equivalence of the result of this algorithm and that of the Tarjan-Vishkin algorithm [35].

Because the number of biconnected components is at most n, the spanning tree, vertex labels, and component heads require only linear space. Therefore, the space requirement of the BC labeling is $O(n)$.

Query on BC labeling. We now show that queries are easy with the BC labeling. An edge is a bridge if and only if it is the only edge connecting a single-vertex component and its component head (the biconnected component contains this single edge). The root of the spanning tree is an articulation point if and only if it is the head of at least two biconnected components. Any other vertex is an articulation point if and only if it is the head of at least one biconnected component.

This new representation can be interpreted as an implicit version of the standard output [17], [25] of biconnected components, i.e. the label of the biconnected component of each edge can be reported in $O(1)$ operations. This is simple: we report the label of the endpoint of the edge that is further from the root along the spanning tree. The correctness can be shown in two cases: if the edge is a spanning tree edge, then the component label is stored in the further vertex; otherwise, the two vertices must have the same label and reporting either one gives the label of this biconnected component.

Using BC labeling gives the following theorem (see Section V-D for the depth analysis):

Theorem V.2. *Articulation points, bridges, and biconnected components on the Asymmetric NP model take $O(m + n\omega)$ expected work and $O(\omega \min\{\omega, m/n\} \log^2 n)$ depth whp, and each query can be answered in $O(1)$ work.*

It is interesting to point out that the BC labeling can efficiently answer queries that are non-trivial when using the standard output. More details about BC labeling can be found in the full version of this paper [7].

C. Biconnectivity Oracle in Sublinear Writes

Next we will show how the implicit k-decomposition generated by Algorithm 1 can be integrated into the algorithm to further reduce writes in the case of bounded-degree graphs. Our goal is as follows.

Theorem V.3. *There exists an algorithm that computes articulation points, bridges, and biconnected components of a bounded-degree graph in $O(n\sqrt{\omega})$ expected work, $O(n/\sqrt{\omega})$ writes and $O(\omega^{3/2} \log^3 n)$ depth, and each query takes an expected $O(\omega)$ work and $O(\omega \log n)$ work whp, with no writes, on the Asymmetric NP model.*

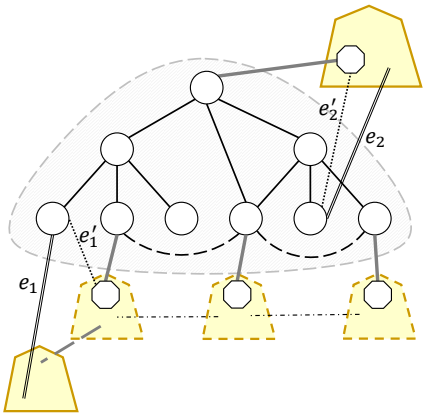

Fig. 3: An example of a local graph. The vertices in the grey shaded area are in one cluster. The local graph contains the vertices in the shaded area and the outside vertices shown in octagons. Solid lines indicate the edges that are in the clusters and thick grey lines represent cluster tree edges connecting other clusters (which are shown in yellow pentagons). The three neighbor clusters sharing the same cluster label are connected using two edges (dashed curves). Edges e_1 and e_2 are the edges that only have one endpoint in the cluster. The other endpoint is set to be the outside vertex connecting the cluster of the other original endpoint of this edge in the cluster spanning tree. Consequently e_1' and e_2' are the replacement edges for e_1 and e_2.

The overall idea of the new algorithm is to replace the vertices in the original graph with the clusters generated by Algorithm 1. We generate the BC labeling on the clusters graph (so the vertex labels are now the ***cluster labels***), and then show that biconnectivity queries can be answered using only the information on the clusters graph and a constant number of associated clusters. The cost analysis is based on the parameter k, and using $k = \sqrt{\omega}$ gives the result in the theorem.

1) The BC labeling on the clusters graph: In the first step of the algorithm we generate the BC labeling on the clusters graph with $k = \sqrt{\omega}$. We root this spanning tree and name it the clusters spanning tree. The head vertex of a cluster is chosen as the ***cluster root*** for that cluster. (The root cluster does not have a cluster root.) For a cluster, we call the endpoint of a cluster tree edge outside of the cluster an ***outside vertex***. The ***outside vertices*** of a cluster is the set of outside vertices of all associated cluster tree edges. Note that all outside vertices except for one are the cluster roots for neighbor clusters.

2) The local graph of a cluster: We next define the concept of the local graph of a cluster, so that each query only needs to look up a constant number of associated local graphs. An example of a local graph is shown in Figure 3 and a more formal definition is as follows.

Definition 4. *The* **local graph** *G' of a cluster is defined as $(V_i \cup V_o, E')$, where V_i is the set of vertices in the cluster, V_o is the set of outside vertices, and E' consists of:*

1) *The edges with both endpoints in this cluster and the associated clusters' tree edges.*
2) *For c neighbor clusters sharing the same cluster label, we find the c corresponding outside vertices in V_o, and connect the vertices with $c - 1$ edges.*

Algorithm 2: Sublinear-write algorithm for biconnectivity

Input: Connected bounded-degree graph $G = (V, E)$ and an implicit k-decomposition.

1 Apply connectivity algorithm to generate the clusters graph.
2 Compute *low*$(\cdot)$ and *high*$(\cdot)$ values of all clusters.
3 Compute the BC labeling of the clusters graph.
// Bridges and articulation points can be queried
4 Compute the root biconnectivity of all outside vertices in all local graphs.
5 Apply leaffix to identify the articulation point of each cluster root.
// Biconnectivity and 1-edge connectivity on vertices and edges can be queried
6 Compute the number of biconnected components in each cluster that are completely within this cluster.
7 Apply prefix sums on the clusters to give an identical label to each biconnected component.
// The label of biconnected component can be queried

3) For an edge (v_1, v_2) with only one endpoint v_1 in V_i, we find the outside vertex v_o that is connected to v_2 on the cluster spanning tree, and create an edge from v_1 to v_o.

Figure 3 shows an example local graph. Solid black lines are edges within the cluster and solid grey lines are cluster tree edges. Neighbor clusters that share a label are shown with dashed outlines and connected via curved dashed lines. e_1 and e_2 are examples of edges with only one endpoint in the cluster, and they are replaced by e_1' and e_2' respectively.

Computing local graphs requires a spanning tree and BC labeling of the clusters graph.

Lemma V.4. *The cost to construct one local graph is $O(k^2)$ in expectation and $O(k^2 \log n)$ whp on the Asymmetric RAM model.*

The analysis of the cost is provided in the full version [7].

3) Queries: With the local graph and the BC labeling on the clusters graph, many types of biconnectivity queries can be answered. The required preprocessing steps are shown in Algorithm 2.

Bridges. There are three cases when deciding whether an edge is a bridge: a tree edge in the clusters spanning tree, a cross edge in the clusters spanning tree, or an edge with both endpoints in the same cluster. Deciding which case to use takes constant operations.

A tree edge is a bridge if and only if it is a bridge of the clusters graph, which we can mark with $O(n/k)$ writes while computing the BC labeling. A cross edge cannot be a bridge.

For an edge within a cluster, we use the following lemma:

Lemma V.5. *An edge with both endpoints in one cluster is a bridge if and only if it is a bridge in the local graph of the corresponding cluster.*

Proof. If an edge is a bridge in the original graph it means that there are no edges from the subtree of the lower vertex to the outside except for this edge itself. By applying the modifications of the edges, this property still holds, which means the edge is still a bridge in the local graph and vice versa. □

Checking if an edge in a cluster is a bridge takes $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*.

Articulation points. By a similar argument, a vertex is an articulation point of the original graph if and only if it is an articulation point of the associated local graph. Given a query vertex v, we can check whether it is an articulation point in the local graph associated to v, which costs $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*.

We now discuss how to perform several more complex queries. To start with, we show some definitions and results that are used in the query algorithms.

Definition 5. *We say a vertex v in a cluster C's local graph is* **root-biconnected** *if v and the cluster root have the same vertex label in C's local graph.*

A root-biconnected vertex v indicates that v can connect to the ancestor clusters without using the cluster root (i.e., the cluster root is not an articulation point to cut v). Another interpretation is that there is no articulation point in the cluster C that disconnects v from the outside vertex of the cluster root.

Lemma V.6. *Computing and storing the root biconnectivity of all outside vertices in all local graphs takes $O(nk)$ operations in expectation and $O(n/k)$ writes on the Asymmetric RAM.*

The proof is straightforward. The cost to construct the local graphs and compute root biconnectivity is $O(nk)$, and because there are $O(n/k)$ clusters tree edges, storing the results requires $O(n/k)$ writes.

Querying whether two vertices are biconnected. Checking whether two vertices v_1 and v_2 can be disconnected by removing any single vertex in the graph is one of the most commonly-used biconnectivity-type queries. To answer this query, our goal is to find the tree path between this pair of vertices and check whether there is an articulation point on this path that disconnects them.

The simple case is when v_1 and v_2 are within the same cluster. We know that the two vertices are connected by a path via the vertices within the cluster. We can check whether any vertex on the path disconnects these two vertices using their vertex labels.

Otherwise, v_1 and v_2 are in different clusters C_1 and C_2. Assume C_{LCA} is the cluster that contains the LCA of v_1 and v_2 (which can be computed by the LCA of C_1 and C_2 in O(1) operations) and $v_{LCA} \in C_{LCA}$ is the LCA vertex. The tree path between v_1 and v_2 is from v_1 to C_1's cluster root, and then to the cluster root of the outside vertex of C_1's cluster root, and so on, until reaching v_{LCA}, and the other half of the path can be constructed symmetrically. It takes $O(k^2)$ expected operations to check whether any articulation point disconnects the paths in C_1, C_2 and C_{LCA}. For the rest of the clusters, because we have already precomputed and stored the root biconnectivity of all outside vertices, then applying a leaffix on the clusters spanning tree computes the cluster containing the articulation point of each cluster root. Therefore, checking whether such an articulation point exists on the path between

C_1 and C_{LCA} or between C_2 and C_{LCA} that disconnects v_1 and v_2 takes $O(1)$ operations. Thus, checking whether two vertices are biconnected requires $O(k^2)$ operations in expectation.

In the full version of this paper [7] we show how to compute whether two vertices are 1-edge connected, biconnected-component labels for edges, bridge-block trees, cut-block trees, and 1-edge-connected components. Each of these queries requires $O(k^2)$ cost in expectation and $O(k^2 \log n)$ *whp* after a precomputation of $O(nk)$ operations and $O(n/k)$ writes. Some of them use more sophisticated techniques.

D. Parallelizing Biconnectivity Algorithms

The two biconnectivity algorithms discussed in this section are highly parallelizable. The key algorithmic components include Euler-tour construction, tree contraction, graph connectivity, prefix sum, and preprocessing LCA queries on the spanning tree. Because the algorithms run each of the components a constant number of times, the depth of the algorithm is bounded by the depth of graph connectivity, whose bound is provided in Section IV ($O(\omega^2 \log^2 n)$ and $O(\omega^{3/2} \log^3 n)$ *whp* when plugging in β as $1/\omega$ and $1/\sqrt{\omega}$, respectively).[5]

For the sublinear-write algorithm, we assume that computations within a cluster are sequential, and the work is upper bounded by $O(k^2) = O(\omega)$ in expectation and $O(k^2 \log n) = O(\omega \log n)$ *whp* for the computations within a cluster. This term is additive to the overall depth, because after computing the spanning tree (forest) of the clusters, we run all computations within the clusters in parallel and then run tree contraction and prefix sums based on the calculated values. The $O(\omega)$ expected work ($O(\omega \log n)$ *whp*) is also the cost for a single biconnectivity query, and multiple queries can be done in parallel.

VI. Sublinear-Write Algorithms on Unbounded-Degree Graphs

Given a graph G with n vertices and m edges that is not bounded-degree, we can construct a bounded-degree graph G' with $O(m+n)$ vertices and edges such that connectivity queries on G can be answered using G', as follows.

The overall idea is to build a tree structure with ***virtual nodes*** for each vertex that has more than a constant degree. Each virtual node will represent a certain range of the edge list. Consider a vertex v_0 with degree d. We name v_0's neighbors v_1 to v_d. We build a binary tree structure with 2 virtual nodes on the first level $v_{0,1\to d/2}$, $v_{0,d/2+1\to d}$, 4 virtual nodes on the second level $v_{0,1\to d/4}, \cdots, v_{0,3d/4+1\to d}$ and so on. We replace the endpoint of an edge from the original graph G with the leaf node in this tree structure that represents the corresponding range with a constant number of edges. Notice that if both endpoints of an edge have large degrees, then they both have to be redirected.

[5]For both algorithms, the classic parallel algorithms with polylogarithmic depth solve Euler-tour construction, tree contraction, and prefix sum, because we here only require linear writes (in terms of the number of vertices, $O(n)$ and $O(n/k)$, for the two algorithms).

The simple case is for a sparse graph in which most of the vertices are bounded-degree, and the sum of the degrees for vertices with more than a constant number of edges is $O(n/k)$ (or $O(n/\sqrt{\omega})$). In this case we can simply explicitly build a tree structure for the edges of a vertex.

Otherwise, we require the adjacency array of the input graph to have the following property: each edge can query its positions in the edge lists for both endpoints. Namely, an edge (u, v) knows it is the i-th edge in u's edge list and j-th edge in v's edge list. To achieve this, either an extra pointer is stored for each edge, or the edge lists are presorted and the label can be binary searched (this requires $O(\log n)$ work for each edge lookup). With this information, there exists an implicit graph G' with bounded-degree. The binary tree structures can be defined such that given an internal tree node, we can find the three neighbors (two neighbors for the root) without explicitly storing the newly added vertices and edges. Notice that the new graph G' now has $O(m+n)$ vertices including the virtual ones. The virtual nodes help to generate implicit k-decomposition and require no writes unless they are selected to be either primary or secondary centers.

Graph connectivity is obviously not affected by this transformation. It is easy to check that a bridge in the original graph G is also a bridge in the new graph G' and vice versa. In the biconnectivity algorithm, an edge in G can be split into multiple edges in G', but this will not change the biconnectivity property within a biconnected component, unless the component only contains one bridge edge, which can be checked separately.

This construction, combined with our earlier results, leads to Theorem I.2.

VII. Conclusion

This work provides several algorithms targeted at solving graph connectivity problems in memories exhibiting read-write asymmetry. Our algorithms make use of an implicit decomposition technique that we believe has wider applications. By using this decomposition, we are able to reduce the number of writes in exchange for a small increase in the total number of operations. This allows us to offset the increased cost of writes in anticipated future systems and hence improve overall performance. In addition to emerging memory technologies, we believe that research into algorithms with fewer writes provides interesting results from both a theoretical and memory/cache coherence perspective. Our work provides a framework that can be used to develop write-efficient solutions to graph connectivity problems.

Acknowledgments

This research was supported in part by NSF grants CCF-1314590, CCF-1314633, CCF-1718700, and CCF-1533858, the Intel Science and Technology Centers for Cloud Computing and Visual Cloud Systems, and the Miller Institute for Basic Research in Science at UC Berkeley.

REFERENCES

[1] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 120–124, 2004.

[2] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proc. ACM SIGMOD*, pages 1753–1758, 2017.

[3] Baruch Awerbuch, Michael Luby, Andrew V. Goldberg, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. IEEE Foundations of Computer Science (FOCS)*, pages 364–369, 1989.

[4] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. Making cost-based query optimization asymmetry-aware. In *Proc. ACM DaMoN*, 2012.

[5] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *European Symposium on Algorithms (ESA)*, 2006.

[6] 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)*, 2016.

[7] 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. *arXiv preprint arXiv:1710.02637*, 2017.

[8] Petra Berenbrink, Bruce Krayenhoff, and Frederk Mallmann-Trenn. Estimating the number of connected components in sublinear time. *Information Processing Letters*, 114(11), 2014.

[9] 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)*, 2015.

[10] 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.

[11] 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.

[12] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6), 2005.

[13] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[14] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endowment*, 8(7):786–797, 2015.

[15] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[16] Richard Cole, Philip N. Klein, and Robert Endre Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 243–250, 1996.

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

[18] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Pawel Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *ACM International Symposium on Experimental Algorithms (SEA)*, 2014.

[19] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005.

[20] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, December 1991.

[21] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996.

[22] Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, pages 1740–1759, 2000.

[23] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symposium on Theory of Computing (STOC)*, 1981.

[24] Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[25] Joseph Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[26] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009.

[27] Nathan Linial and Michael E. Saks. Decomposing graphs into regions of small diameter. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 91, pages 320–330, 1991.

[28] Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proc. 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013.

[29] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proc. ACM SIGMOD*, pages 371–386, 2016.

[30] Hyoungmin Park and Kyuseok Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8), 2009.

[31] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[32] Chung Keung Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *Proc. ISAAC*, pages 212–222, 1997.

[33] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 143–153, 2014.

[34] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Physical review letters*, 58(2):86, 1987.

[35] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[36] Stratis D. Viglas. Adapting the B^+-tree for asymmetric I/O. In *East European Conference on Advances in Databases and Information Systems (ADBIS)*, 2012.

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

[38] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A Jiménez. WADE: Writeback-aware dynamic cache management for NVM-based main memory system. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):51, 2013.

[39] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007.

[40] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):53, 2012.

[41] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009.

[42] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013.