Implicit Decomposition for Write-Efficient Connectivity Algorithms

Naama Ben-David[1], Guy E. Blelloch[1], Jeremy T. Fineman[2], Phillip B. Gibbons[1], Yan Gu[1], Charles McGuffey[1], and Julian Shun[3]

1 Carnegie Mellon University
2 Georgetown University
3 University of California, Berkeley

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, 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, we also provide the first $o(m)$ writes and $O(m)$ operations parallel algorithms for connectivity and biconnectivity. These algorithms provide insight into how applications can efficiently process computations on large graphs in systems with read-write asymmetry.

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.

1 Introduction

The future of main memory appears to lie in a new wave of nonvolatile memory technologies (e.g., phase-change memory, spin-torque transfer magnetic RAM, memristor-based resistive RAM, conductive-bridging RAM) that promise persistance, significantly lower energy costs, and higher density than the DRAM technology used in today's main memories [20, 23, 28, 43]. A key property of such technologies, however, is their asymmetric read-write costs: writes can be an order of magnitude or more higher energy, higher latency, lower (per-module) bandwidth, and more wear-out prone than reads [1, 2, 7, 6, 8, 10, 13, 14, 21, 22, 26, 27, 33, 41, 42, 44, 45]. Moreover, because bits are stored in these technologies as "at rest" states of the given material that can be quickly read but require physical change to update, this asymmetry appears fundamental.[1] This motivates the need for algorithms that are *write-efficient*, in that they significantly reduce the number of writes compared to existing algorithms.

Models of computation for memories with asymmetric read-write costs have been proposed and studied by a number of recent papers [3, 15, 16, 30, 9, 39, 40, 6, 7, 4, 8, 24] (see Appendix B for details). In this paper, we focus on two such models that are simple enough for algorithm design while still capturing the read-write asymmetry: (i) the *Asymmetric RAM model* [7], 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 [4]. Both models have a small symmetric memory that can be used to help minimize the number of writes to the large asymmetric memory.

[1] See Appendix A for more technical details of the new memories.

© N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey and J. Shun;
licensed under Creative Commons License CC-BY
25th Annual European Symposium on Algorithms (ESA 2017).

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

	Connectivity		Biconnectivity		Best choice
	Seq. time	Parallel work	Seq. time	Parallel work	when
Prior work	$O(m+\omega n)$	$O(\omega m)^\dagger$	$O(\omega m)$	$O(\omega m)^\dagger$	–
Ours [§4.2, §5.2]	–	$O(m+\omega n)^\dagger$	$O(m+\omega n)$	$O(m+\omega n)^\dagger$	$m \in \Omega(\sqrt{\omega}n)$
Ours [§4.3, §5.3]	$O(\sqrt{w}m)^\dagger$	$O(\sqrt{w}m)^\dagger$	$O(\sqrt{w}m)^\dagger$	$O(\sqrt{w}m)^\dagger$	$m \in o(\sqrt{\omega}n)$

Table 1 Summary of our main results (n nodes, m edges, †=expected), where $\omega \gg 1$ is the cost of writes to the asymmetric memory. Query times are $O(\sqrt{\omega})^\dagger$ (connectivity) and $O(\omega)^\dagger$ (biconnectivity) for the last row and $O(1)$ for the rest. All parallel algorithms are low depth.

We present the first write-efficient sequential and parallel algorithms for graph connectivity (connected components, spanning forests) and biconnectivity (biconnected components, articulation points, and related 1-edge-connectivity) problems. The algorithms significantly reduce the number of writes to the asymmetric memory compared to existing algorithms, improving the overall sequential time and parallel work bounds. In the algorithms we often cannot afford to write out the result since just writing it requires too many writes. Instead we construct a compact *"oracle"* that can answer queries (e.g. if two vertices are connected) efficiently. The costs for constructing the oracles and query times are summarized in Table 1.

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 [36, 11, 17, 32, 31, 19, 18], and hence are actually $\Theta(\omega m)$ work in the asymmetric memory setting. We show how the algorithm of Shun et al. [36] can be adapted to use only $O(m/\omega+n)$ writes (and $O(m)$ other operations), by avoiding repeated graph contractions and using a recent write-efficient BFS [4], 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 [12, 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 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 constant work. Sequentially, the construction takes $O(m + \omega n)$ time on the Asymmetric RAM model, with constant query time.*

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.

[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).

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. Appendix H 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 by only looking at 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 contains the respective query points. However, biconnectivity queries require considerable subtleties in the design, to store the appropriate information on the clusters graph that enables each query to access only a constant number of clusters (at most 3).

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 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 in $O(\omega)$ expected time (work) ($O(\omega \log n)$* whp*).*

2 Preliminaries

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*** query on a tree (as summarized in Appendix D). Let $[n] = \{1, 2, \cdots, n\}$ where n is a positive integer.

Computation models. Sequential algorithms are analyzed using the ***Asymmetric RAM*** model [7], 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 have unit cost. This models practical settings in which there is a small amount of standard symmetric memory (e.g., DRAM) in addition to the asymmetric memory.

For parallel algorithms, we use the ***Asymmetric Nested-Parallel (NP)*** model [4], 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. [4] showed that a work-stealing scheduler can execute an algorithm with work W and depth D on the Asymmetric NP Model 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 any writes to symmetric memory. All reads and writes are to words of size $\Theta(\log n)$ for input size n. The size of the symmetric memory is measured in words.

3 Implicit Decomposition

In this paper we introduce the concept of an implicit decomposition. The idea is to partition a 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 only undirected graphs.

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*** $\{v \in V \mid \rho(v) = s\}$ 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 $|S| = O(n/k)$ (i.e. all clusters are about the same size). We are often interested in the graph induced by the decomposition, and in particular:

▶ **Definition 3** (clusters graph). 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 4** (implicit decomposition). An ***implicit decomposition*** of a connected graph $G = (V, E)$ is a decomposition (S, ρ) such that $\rho(\cdot)$ is defined implicitly in terms of an algorithm given only G, S, and a labeling on 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 5.** *An implicit k-decomposition can be built on a 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;*
- *the labels are 1-bit per vertex;*
- *$\rho(v)$ requires $O(k)$ operations in expectation and $O(k \log n)$* whp*, and no writes;*
- *finding all vertices in a cluster given its center takes $O(k^2)$ operations in expectation, and $O(k^2 \log n)$ operations* whp *and no writes; and,*
- *construction and queries can be done in $O(k \log n)$ symmetric memory* whp.

At a high level, we identify 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). Our construction is based on first selecting 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, we identify 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 we store are the set S and a bit representing whether each center is primary or not.

It turns out to be very 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. We use the notation $SP(u, v) + SP(v, w)$ to indicate joining the two shortest paths at v. 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. $\rho(v)$ 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 every vertex 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.

Algorithm 1: Constructing k-Implicit Decomposition

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

```
1  Sample each vertex with probability 1/k, and place in S_0
2  S_1 = ∅
3  foreach vertex v ∈ S_0 do
4  |  SecondaryCenters(v, G, S_0)
5  return S_0 and S_1
6  function SecondaryCenters(v, G, S)
7  |  Search from v for the first k vertices that have v as their center. This defines a tree.
8  |  If the search exhausts all vertices with center v, return.
9  |  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.
10 |  Add u to S_1.
11 |  SecondaryCenters(v, G, S ∪ u)
12 |  SecondaryCenters(u, G, S ∪ u)
```

▶ **Lemma 6.** *$\rho(v)$ 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. Since 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 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*. ◀

For any center $v \in S$, finding its cluster $C(v)$ is easy; we simply run a BFS starting at v. For each vertex $u \in V$ that the algorithm visits, it checks if $\rho(u) = v$. If so, we add u to $C(v)$ and put its unvisited neighbors in the BFS queue. We claim that this algorithm finds the cluster. To show this, we use the following lemma, whose proof is in Appendix E.

▶ **Lemma 7.** *The shortest paths used to define $\rho(v)$ define a rooted spanning tree on each cluster, with the center as the root.*

▶ **Corollary 8.** *Any vertex u for which $\rho(u) = v$ has a path to v contained in v's cluster.*

By Corollary 8, the BFS from v will visit all vertices in $C(v)$. Furthermore, since the graph has bounded degree, it will only visit $O(C(v))$ vertices not in $C(v)$. Each visit to a vertex u requires finding $\rho(u)$. By Lemma 6, each of these takes $O(k)$ operations in expectation, and $O(k \log n)$ operations *whp*. We only need space for storing the queue and $C(v)$ (both of size $O(|C(v)|)$), and for each search ($O(k \log n)$ *whp*). Thus, we have the following lemma.

▶ **Lemma 9.** *For any vertex $v \in S$, its cluster $C(v) = \{u \in V \mid \rho(u) = v\}$ can be found in $O(k|C(v)|)$ operations in expectation and $O(k|C(v)| \log n)$ operations* whp*, and using $O(|C(v)| + k \log n)$ symmetric memory* whp.

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 7, 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 10.** *Algorithm 1 runs in $O(nk)$ operations and $O(n/k)$ writes (both in expectation), and $O(k \log n)$ symmetric memory* whp *on the Asymmetric RAM Model. It generates a k-implicit 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 since 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* since 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 can throw out the symmetric memory and write out u to S_1, requiring one write per call to SecondaryCenters.

We are left with showing there are at most $O(n/k)$ calls to SecondaryCenters. There are n/k primary clusters in expectation. If there are too many (beyond some constant factor above the expectation), we can try again. Since 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 [34]. We can now count all internal nodes of the recursion against the leaves. There are at most $O(n/k)$ leaves since each defines a cluster of size $\Theta(k)$. Therefore there are $O(n/k)$ calls to SecondaryCenters, giving the overall bounds stated. ◀

Parallelizing the decomposition. To parallelize the decomposition in Algorithm 1, we make one small change; in addition to adding the secondary cluster u at each recursive call to SecondaryCenters, we add all children of v. 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, whose proof we defer to Appendix E.

▶ **Lemma 11.** *Algorithm 1 runs in depth $O(k \log n(k^2 \log n + \omega))$ on the Asymmetric NP.*

Extension to unconnected graphs. In the above discussion, we assumed the input graph is connected. However, for some problems, like graph connectivity, the graph is not necessarily connected. In Appendix E, we show how to extend the definition of implicit k-decomposition and the algorithm to generate the cluster centers for unconnected input graphs.

4 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 task 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 [36, 11, 17, 32, 31, 19, 18], all of them use $\Omega(n + m)$ writes. This section has two main contributions: (1) Section 4.2 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 4.3 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 4.2) can be viewed as a write-efficient version of the parallel algorithm due to Shun et al. [36]. This algorithm uses a low-diameter decomposition

algorithm of Miller et al. [29] as a subroutine, which we review and adapt next in Section 4.1 and Appendix F. Any omitted proofs for this section appear in Appendix F.

4.1 Low-diameter Decomposition

Here we review the low-diameter decomposition of Miller et al. [29]. 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 only important to extract a bound on the number of writes, and it is briefly summarized in Appendix F.

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 $V_1, \ldots, V_k$ such that (1) the shortest path between any two vertices in each V_i using only vertices in V_i is 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. [29] provide an efficient parallel algorithm for generating a $(\beta, O(\log n/\beta))$-decomposition. 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 by write-efficient BFS's [4] yields the following theorem:

▶ **Theorem 12.** *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.*

4.2 Connectivity and Spanning Forest

The parallel connectivity algorithm of [36] 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 [4].
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 [4].
4. Run any parallel linear-work spanning forest algorithm on the contracted graph, e.g., the algorithm from [11] 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 13.** *For any choice of $0 < \beta < 1$, connectivity and spanning forest can be solved in $O(n + \beta m)$ expected writes, $O(\omega n + \beta \omega m)$ expected work, and $O(\omega \log^2 n/\beta)$ depth* whp *on the Asymmetric NP model. For $\beta = 1/\omega$, these bounds reduce to $O(n + m/\omega)$ expected writes, $O(m + \omega n)$ expected work and $O(\omega^2 \log^2 n)$ depth* whp.

4.3 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 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 6.

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 a k-implicit decomposition of the graph.
2. Run the write-efficient connectivity algorithm from Section 4.2 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 adjacent to a center vertex x in the clusters graph. 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 9) — 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 14.** *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.

Note that a consequence of the implicit representation is that listing neighbors is more expensive, and thus the number of operations performed by BFS increases by an $O(k^2)$ factor, 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.

▶ **Theorem 15.** *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 algorithm with sublinear writes.

5 Biconnectivity

In this section we introduce algorithms related to biconnectivity and 1-edge connectivity queries. Due to the page limits, here we give an extended abstract of the results while the full version is provided in Appendix G.

We first review the classic approach, in which the challenge arises since just the output requires $O(m)$ writes. Then we propose a new BC (biconnected-component) labeling as 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 explaining sequential algorithms that we believe to be new and interesting. Then in Section 5.4 we show that these algorithms are parallelizable. For this section, we assume the size of the symmetric memory in our model is $O(k \log n)$.

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.

5.1 The Classic Algorithm

The classic parallel algorithm [38] 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 $\mathit{first}(v)$ and $\mathit{last}(v)$, which are the ranks of v's first and last appearance on the Euler tour of T. The algorithm also computes $\mathit{low}(v)$ and $\mathit{high}(v)$ which indicate the first and last vertex on the Euler tour that are connected by a nontree edge to the subtree rooted at v. The $\mathit{low}(\cdot)$ and $\mathit{high}(\cdot)$ values can be computed by a reduce on each vertex followed by a leaffix on the subtrees. The computation takes $O(\omega \log n)$ depth, $O(m)$ work, and $O(n)$ writes on the Asymmetric NP model, by using the algorithm and scheduling theorem in [4]. 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* of the parent.

The standard output of biconnected components [12, 25] is an array $B[\cdot]$ with size m, where the i-th element in B indicates which biconnected component the i-th edge belongs to. Explicitly writing-out B is costly in the asymmetric setting, especially when $m \gg n$. We provide an alternative BC labeling as output that only requires $O(n)$ writes.

5.2 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. We will later show how to use this representation along with an implicit decomposition to reduce the writes further.

▶ **Definition 16** (BC labeling). The BC labeling of a connected graph with respect to a rooted spanning tree stores a ***vertex label*** $l : V \backslash \{\mathit{root}\} \to [C]$ where C is the number of biconnected components in the graph, and a ***component head*** $r : [C] \to V$ of each biconnected component.

▶ **Lemma 17.** *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 about 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 iff there exists at least one child u in the spanning tree that has $\mathit{first}(v) \leq \mathit{low}(u)$ and $\mathit{high}(u) \leq \mathit{last}(v)$, and here we name the tree edge between such a pair of vertices to be 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 4.2 gives a unique component label that we assign as the vertex label. For each component, its head is the vertex that is the parent of its cluster in the spanning tree. 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 [38].

Since 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 BC labeling is $O(n)$.

Biconnectivity and related 1-edge-connectivity queries on BC labeling are easy, and we review them in Appendix G.2.

The BC labeling gives the biconnectivity part of Theorem 1, which is formally stated as Theorem 18 in Appendix G (see Section 5.4 for depth analysis).

5.3 Biconnectivity Oracle in Sublinear Writes

Next we will show how the implicit k-decomposition generated by Algorithm 1 can be integrated into the biconnectivity algorithm to further reduce writes. In this section we show the biconnectivity part of Theorem 2 in the case of bounded-degree graphs. A more formal result is stated as Theorem 19 in Appendix G.

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 a connected-type query 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.

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 a ***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.

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 2 and a more formal definition is as follows.

▶ **Definition 20** (local graph). The local graph G' of a cluster is defined as $(V_i \cup V_o, E')$. V_i is the set of vertices in the cluster and V_o is the set of outside vertices. 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.
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 2 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 a local graphs requires a spanning tree and BC labeling of the clusters graph.

▶ **Lemma 21.** *The cost to construct one local graph is $O(k^2)$ in expectation and $O(k^2 \log n)$* whp *on the Asymmetric RAM.*

Queries. With the definition of the local graph and the BC labeling on the clusters graph, biconnectivity queries can be answered. Here we take bridges as an example, and the queries for articulation points, **whether two vertices are biconnected**, **whether two vertices are 1-edge connected**, and **biconnected-component labels for edges** are shown in In Appendix G.3. The preprocess steps for the queries are shown in an overview of Algorithm 2. With the concepts and lemmas in this section, with a precomputation of $O(nk)$ cost and $O(n/k)$ writes, we can also do a normal query with $O(k^2)$ cost in expectation and $O(k^2 \log n)$ *whp* on **bridge-block tree**, **cut-block tree**, and **1-edge-connected components**.

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 22.** *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 the corresponding cluster.*

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

5.4 Parallelizing Biconnectivity Algorithms

The two biconnectivity algorithms discussed in this section are essentially highly parallelizable. The key algorithmic components include Euler-tour construction, tree contraction, graph connectivity, prefix sum, and preprocessing LCA queries on the spanning tree. Since the algorithms run each of the components a constant number of times, and the depth of the algorithm is bounded by the depth of graph connectivity, whose bound is provided in Section 4 ($O(\omega^2 \log^2 n)$ and $O(\omega^{3/2} \log^3 n)$ *whp* respectively when plugging in β as $1/\omega$ and $1/\sqrt{\omega}$).[3]

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 any computations within a cluster. This term is additive to the overall depth, since after acquiring 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 ones can be queried in parallel.

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

References

1. Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajech K. Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.
2. Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path processing using solid state storage. In *Proc. International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2012.
3. Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *Proc. European Symposium on Algorithms (ESA)*, 2006.
4. 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 *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.
5. Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.
6. Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.
7. Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *24th Annual European Symposium on Algorithms*, pages 14:1–14:18, 2016.
8. 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*, pages 648–658, 2016.
9. Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2011.
10. Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
11. Richard Cole, Philip N. Klein, and Robert Endre Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *SPAA*, pages 243–250, 1996.
12. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.
13. Xiangyu Dong, Norman P. Jouupi, and Yuan Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *Proc. ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.
14. Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai H. Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Proc. ACM Design Automation Conference (DAC)*, 2008.
15. 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 *Proc. ACM International Symposium on Experimental Algorithms (SEA)*, 2014.
16. Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005.
17. Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, December 1991.
18. 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.

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

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

21 Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, Shouzhen Gu, and Edwin Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014.

22 www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014.

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

24 Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, 2017.

25 J. Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

26 Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2014.

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

28 Jagan S. Meena, Simon M. Sze, Umesh Chand, and Tseung-Yuan Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014.

29 Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the ACM symposium on Parallelism in Algorithms and Architectures*, pages 196–203, 2013.

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 *ISAAC*, pages 212–222, 1997.

33 Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems.* Morgan & Claypool, 2011.

34 A.L. Rosenberg and L.S. Heath. *Graph Separators, with Applications.* Frontiers in Computer Science. Springer US, 2001.

35 Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *International Symposium on Algorithms and Computation*, pages 14–24. Springer, 2002.

36 Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 143–153, 2014.

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

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

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

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

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 leafix 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

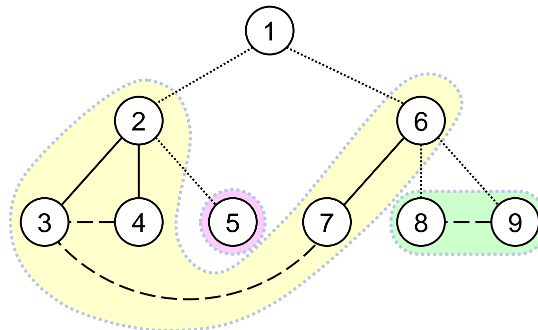

Figure 1 An example of the BC labeling of a graph. The spanning tree is rooted at vertex 1. The solid and dot lines indicate tree edges while dot lines are the critical edges. Dash lines are non-tree edges. The vertex labels $l = [1, 1, 1, 2, 1, 1, 3, 3]$, and component heads $r = [1, 2, 6]$. Based on the BC labeling the bridges, articulation points, and biconnected components can be easily retrieved as $\{(2, 5)\}$, $\{2, 6\}$, and $\{\{1, 2, 3, 4, 6, 7\}, \{2, 5\}, \{6, 8, 9\}\}$.

41 Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *Proc. IEEE Design, Automation and Test in Europe (DATE)*, 2011.

42 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 *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007.

43 Yole Developpement. Emerging non-volatile memory technologies, 2013.

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

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

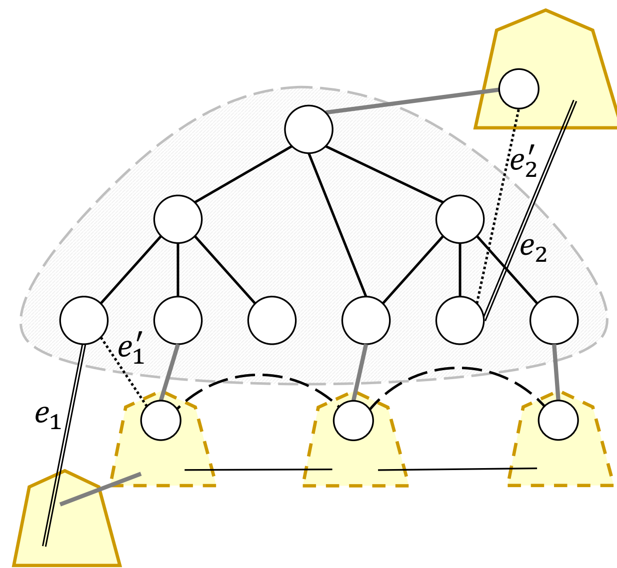

Figure 2 An example of a local graph. The vertices in the shaded area is in one cluster. The local graph contains the vertices in the shaded area and the outside vertices shown in smaller circles. Solid lines indicate the edges that are in the clusters and thick grey lines represent cluster tree edges connecting other clusters (which are shown in yellow pentagons). The three neighbor clusters sharing the same cluster label are connected using two edges (dash curves). Edges e_1 and e_2 are the edges that only has one endpoints 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 replaced edges for e_1 and e_2.

A Motivation from [6]

Further motivation for the asymmetry between reads and write costs in emerging memory technologies was provided in [6]. As a convenience to the reviewer, in this appendix we repeat a suitable excerpt from that paper.

"While DRAM stores data in capacitors that typically require refreshing every few milliseconds, and hence must be continuously powered, emerging NVM technologies store data as "states" of the given material that require no external power to retain. Energy is required only to read the cell or change its value (i.e., its state). While there is no significant cost difference between reading and writing DRAM (each DRAM read of a location not currently buffered requires a write of the DRAM row being evicted, and hence is also a write), emerging NVMs such as Phase-Change Memory (PCM), Spin-Torque Transfer Magnetic RAM (STT-RAM), and Memristor-based Resistive RAM (ReRAM) each incur significantly higher cost for writing than reading. This large gap seems fundamental to the technologies themselves: to change the physical state of a material requires relatively significant energy for a sufficient duration, whereas reading the current state can be done quickly and, to ensure the state is left unchanged, with low energy. An STT-RAM cell, for example, can be read in 0.14 ns but uses a 10 ns writing pulse duration, using roughly 10^{-15} joules to read versus 10^{-12} joules to write [14] (these are the raw numbers at the materials level). A Memristor ReRAM cell uses a 100 ns write pulse duration, and an 8MB Memrister ReRAM chip is projected to have reads with 1.7 ns latency and 0.2 nJ energy versus writes with 200 ns latency and 25 nJ energy [41]—over two orders of magnitude differences in latency and energy. PCM is the most mature of the three technologies, and early generations are already available as I/O devices. A recent paper [26] reported 6.7 μs latency for a 4KB read and 128 μs latency for a 4KB write. Another reported that the sector I/O latency and bandwidth for random 512B writes was a factor of 15 worse than for reads [22]. As a future memory/cache replacement, a 512Mb PCM memory chip is projected to have 16 ns byte reads versus 416 ns byte writes, and writes to a 16MB PCM L3 cache are projected to be up to 40 times slower and use 17 times more energy than reads [13]. While these numbers are speculative and subject to change as the new technologies emerge over time, there seems to be sufficient evidence that writes will be considerably more costly than reads in these NVMs."

Note that, unlike SSDs and earlier versions of phase-change memory products, these emerging memory products will sit on the processor's memory bus and be accessed at byte granularity via loads and stores (like DRAM). Thus, the time and energy for reading can be roughly on par with DRAM, and depends primarily on the properties of the technology itself relative to DRAM.

B Further Details on Prior Work on Asymmetirc Memory

Read-write asymmetries have been studied in the context of NAND Flash chips [3, 15, 16, 30], focusing on how to balance the writes across the chip to avoid uneven wear-out of locations. Targeting instead the new technologies, Chen et al. [9] and Viglas [39, 40] presented write-efficient algorithms for database operators such as hash joins and sorting. Blelloch et al. [6] defined several sequential and parallel computation models that take asymmetric read-write costs into account, and analyzed and designed sorting algorithms under these models. Their follow-up paper [7] presented sequential algorithms for various problems that do better than their classic counterparts under asymmetric read-write costs, as well as several lower bounds. Carson et al. [8] presented 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. [4] proposed a nested-parallel model with asymmetric read-write costs and presented 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] showed 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.

C Discussion on Write-Efficient Connectivity Algorithms

One may question the necessity of write-efficient connectivity algorithms since it seems that loading the graph to the asymmetric memory requires $O(m)$ space and writes. While this is true in some applications, it is not always the case.

First of all, there are many applications in which the graph is represented implicitly. For example the famous Swendsen-Wang algorithm [37] (for studying ferromagnetic phase transitions) is a subgraph of the 2D or 3D mesh in which an edge exists based on local state, and changes from step to step. Each step requires finding connected components. One need not represent the graph explicitly (and most implementations do not).

Secondly, even if the entire graph needs to be stored in the memory, it is still common to define a subgraph using a predicate on edge or vertex labels, and then query (bi)connectivity on the subgraph. For example, in a typical social network we might query whether two users are connected (via a path). Connections of interest might be limited to the last week or the last month, or use other vertex/edge availability based on the labels or local information. On roadmaps we might filter out roads with different conditions and check (bi)connectivity. By running our algorithm(s) on different subgraphs and storing our new representations, we can efficiently answer queries for multiple different subgraphs in less space and work than the classic approaches.

Last but not the least, this paper is concerned with the theoretical question of a tighter upper bound on computing (bi)connectivity under the Asymmetric RAM model, whether it is possible to compute (bi)connectivity with a sublinear number of writes, and what the tradeoffs are with respect to extra reads. We believe this question is of interest beyond the particulars of new memory technologies.

D Formal Definitions of the Terms

A ***spanning tree*** T of an undirected connected graph G is a subgraph that is a tree which includes all of the vertices of G. A ***spanning forest*** of G contains the union of the spanning trees of all connected components in G. The ***lowest-common-ancestor*** query for two vertices on a rooted spanning tree requires $O(n)$ work and $O(\log n)$ depth on preprocessing, and $O(1)$ query time [5, 35].

A ***connected component*** of G is a subgraph in which any two vertices are connected to each other by paths via edges in the graph.

A ***biconnected component*** (also known as a block or 2-connected component) of G is a maximal subgraph such that it is still connected after removing any single vertex in the subgraph. Any connected graph decomposes into a tree of biconnected components called

the block-cut tree of the graph. The blocks are attached to each other at shared vertices called ***articulation points***.

A ***bridge*** of G is an edge whose deletion increases the number of connected components of the graph. A connected graph is k***-edge-connected*** if it remains connected whenever fewer than k edges are removed. An unconnected graph is 0-edge connected; a connected graph with bridges is 1-edge-connected; and a bridge-less graph is at least 2-edge-connected.

E Proof details for Implicit Decomposition

▶ **Lemma 7.** *The shortest paths used to define $\rho(v)$ define a rooted spanning tree on each cluster, with the center as the root.*

Proof. We first show that this is true for the clusters defined by the primary centers S ($\rho_0(v)$). Consider a vertex v with $\rho_0(v) = s$, and consider all 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 $L(SP(v, s)) = L(SP(v, u) + SP(u, s))$. We know that since $\rho_0(v) = s$ 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 $SP(v, s)$ since $u \in SP(v, s)$. 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.

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 rooted at one of those centers and going down until another center is hit. Each vertex in the tree for s will be assigned the correct partition by $\rho(v)$ since each will be assigned to the first secondary center on the way to the primary center. ◀

▶ **Lemma 11.** *Algorithm 1 runs in depth $O(k \log n(k^2 \log n + \omega))$ on the Asymmetric NP.*

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. In this way one is removed from the height of the tree 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* since by Lemma 6 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. Notice that once a connected component contains at least one primary center, the definition and Theorem 5 hold. However, it is possiblethat 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.

F Additional Details and Proofs for Connected Components and Spanning Forests

F.1 Summary of Low-Diameter Decomposition Algorithm

The algorithm of Miller et al. [29] generates a $(\beta, O(\log n/\beta))$ decomposition with $O(m)$ operations and $O(\omega\log^2 n/\beta)$ depth *whp*. As described by Miller et al., the number of writes performed is also $O(m)$, but this can be improved to $O(n)$. Specifically, the algorithm executes multiple breadth-first searches (BFS's) in parallel, which can be replaced by write-efficient BFS's.

In more detail, the algorithm first assigns each vertex v a value δ_v drawn from an exponential distribution with parameter β (mean $1/\beta$). Then on iteration i of the algorithm, BFS's are started from unexplored vertices v where $\delta_v \in [i, i+1)$ and all BFS's that have already started are advanced one level. At the end of the algorithm, all vertices that were visited by a BFS starting from the same source will belong to the same subset of the decomposition. If a vertex is visited by multiple BFS's in the same iteration, it can be assigned to an arbitrary BFS.[4] The maximum value of δ_v can be shown to be $O(\log n/\beta)$ *whp*, and so the algorithm terminates in $O(\log n/\beta)$ iterations. Each iteration requires $O(\omega \log n)$ depth for packing the frontiers of the BFS's, leading to an overall depth of $O(\omega \log^2 n/\beta)$ *whp*. A standard BFS requires operations and writes that are linear in the number of vertices and edges explored, giving a total work of $O(\omega(m+n))$. By using the write-efficient BFS from [4], the expected number of writes for each BFS is proportional to the number of vertices marked (assigned to it), and so the total expected number of writes is $O(n)$. Tasks only need $O(1)$ symmetric memory in the algorithm. This yields Theorem 12.

F.2 Proofs Omitted from Section 4

Proof of Theorem 13. Step 1 has performance bounds given by Theorem 12, and the expected number of edges remaining in the contracted graph is at most βm. Step 2 performs BFS's on disjoint subgraphs, so summing across subsets yields $O(n)$ expected writes and $O(m + n\omega)$ expected work. Since each tree has low diameter $\mathcal{D} = O(\log n/\beta)$, the BFS's have depth $O(\omega\mathcal{D}\log n) = O(\omega \log^2 n/\beta)$ *whp* [4]. Step 3 is dominated by the filter, which has a number of writes proportional to the output size of $O(\beta m)$, for $O(m + \beta\omega m)$ work. The depth is $O(\omega \log n)$ [4]. Finally, the algorithm used in Step 4 is not write-efficient, but the size of the graph is $O(n + \beta m)$, giving that many writes and $O(\omega(n + m\beta))$ work. Adding these bounds together yields the theorem. ◀

Proof of Lemma 14. Listing all the vertices in the cluster takes expected work $O(k^2)$ according to Lemma 9, or $O(k^2 \log n)$ *whp*. The number of vertices in the cluster is $O(k)$, so

[4] The original analysis of Miller et al. [29] requires the vertex to be assigned to the BFS with the smaller fractional part of δ_s, where s is the source of the BFS. However, Shun et al. [36] show that an arbitrary assignment gives the same complexity bounds.

they can all fit in symmetric memory. Moreover, since 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 (Lemma 6). ◀

Proof of Theorem 15. The k-implicit decomposition can be found in $O(n/k)$ writes, $O(kn + \omega n/k)$ work, and $O(k \log n(k^2 \log n + \omega))$ depth by Lemmas 10 and 11. For $k = \sqrt{\omega}$, these bounds reduce to $O(n/\sqrt{\omega})$ writes, $O(\omega n)$ work, and $O(\omega^{3/2} \log^3 n)$ depth.

If we had an explicit representation of the clusters graph with $n' = O(n/k)$ vertices and $m' = O(m) = O(n)$ edges, the connectivity algorithm would have $O(n' + m'/k) = O(n/k)$ expected writes, $O(\omega n' + \omega m'/k) = O(\omega n/k)$ expected work, and $O(\omega k \log^2 n)$ depth *whp* (by Theorem 13). The fact that the clusters graph is implicit means that the BFS work (but not writes) is multiplied by a $O(k^2)$ factor, giving expected work $O(\omega kn)$. To get a high probability bound, the depth is multiplied by $O(k^2 \log n)$, giving us $O(\omega k^3 \log^3 n)$. For $k = \sqrt{\omega}$, the work and writes match the theorem statement, but the depth is larger than claimed by a ω factor.

To remove the extra ω factor on the depth, we need to look inside the BFS algorithm and its analysis [4]. The $O(\omega \mathcal{D} \log n)$ depth bound for the BFS, where $\mathcal{D} = O(k \log n)$ is the diameter, is dominated by the depth of a packing subroutine on vertices of the frontier. This packing subroutine does not look at edges, and is thus not affected by the overhead of finding a vertex's neighbors in the implicit representation of the clusters graph. Ignoring the packing and just looking at the exploration itself, the depth of BFS is $O(\mathcal{D} \log n)$, which given the implicit representation increases by a $O(k^2 \log n)$ factor. Adding these together, we get depth $O(\omega k \log^2 n + k^3 \log^3 n) = O(\omega^{3/2} \log^3 n)$ for the BFS phases. ◀

G Full Version of Graph Biconnectivity

Here we provide the full version of our algorithms related to biconnectivity and 1-edge connectivity queries. We first review the classic approach and its output, which requires $O(m)$ 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 explaining sequential algorithms that we believe to be new and interesting. Then in Section G.4 we show that these algorithms are parallelizable. For this section, we assume the size of the symmetric memory in our model is $O(k \log n)$.

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.

G.1 The Classic Algorithm

The classic parallel algorithm [38] 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:

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

where

$$w(u) = \min\{first(u) \cup \{first(u') \mid (u, u') \textit{ is a nontree edge}\}\}^5$$

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 on the subtrees. The computation takes $O(\omega \log n)$ depth, $O(m)$ work, and $O(n)$ writes on the Asymmetric NP model, by using the algorithm and scheduling theorem in [4]. 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* of the parent.

The standard output of biconnected components [12, 25] is an array $B[\cdot]$ with size m, where the i-th element in B indicates which biconnected component the i-th edge belongs to. Explicitly writing-out B is costly in the asymmetric setting, especially when $m \gg n$. We provide an alternative BC labeling as output that only requires $O(n)$ writes.

G.2 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. We will later show how to use this representation along with an implicit decomposition to reduce the writes further.

▶ **Definition 16** (BC labeling)**.** The BC labeling of a connected graph with respect to a rooted spanning tree stores a ***vertex label*** $l : V \backslash \{root\} \to [C]$ where C is the number of biconnected components in the graph, and a ***component head*** $r : [C] \to V$ of each biconnected component.

▶ **Lemma 17.** *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 about 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 iff there exists at least one child u in the spanning tree that has $first(v) \le low(u)$ and $high(u) \le last(v)$, and here we name the tree edge between such a pair of vertices to be 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 4.2 gives a unique component label that we assign as the vertex label. For each component, its head is the vertex that is the parent of its cluster in the spanning tree. 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 [38].

Since 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 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 iff it is the only edge connecting a single-vertex component and its component head. The root of the spanning tree is an articulation point iff it is the head of at least two biconnected components. Any other vertex is an articulation point iff it is the head of at least one biconnected component. A block-cut tree can also be generated from the BC labeling: for each vertex create an edge from itself and its vertex label; and for each component create an edge from the label of this component to the component head. We have a block-cut tree after removing degree-1 nodes corresponding to vertices.

This new representation can be interpreted as an implicit version of the standard output [12, 25] of biconnected components, i.e. the label of the biconnected component of each edge can be reported within $O(1)$ 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 G.4 for depth analysis).

▶ **Theorem 18.** *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. For example, consider the query: are two vertices in the same biconnected component? With the BC labeling we can answer the query by finding the label of the lower vertex and checking whether the higher one has the same label or is the component head of this component. To the best of our knowledge, answering such queries on the standard representation can be hard, unless other information is also kept (e.g. a block-cut tree).

G.3 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 19.** *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.*

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 a connected-type query 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.

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 a ***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.

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 2 and a more formal definition is as follows.

▶ **Definition 20** (local graph)**.** The local graph G' of a cluster is defined as $(V_i \cup V_o, E')$. V_i is the set of vertices in the cluster and V_o is the set of outside vertices. 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.
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 2 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 a local graphs requires a spanning tree and BC labeling of the clusters graph.

▶ **Lemma 21.** *The cost to construct one local graph is $O(k^2)$ in expectation and $O(k^2 \log n)$* whp *on the Asymmetric RAM.*

Proof. Each cluster in the implicit k-decomposition has at most k vertices, so finding the vertices V_i takes $O(ck)$ cost where c is the cost to compute the mapping $\rho(\cdot)$ of a vertex ($O(k)$ in expectation and $O(k \log n)$ *whp*). Since each vertex has a constant degree, there will be at most $O(k)$ neighbor clusters, so $|V_o| = O(k)$. Enumerating and checking the other endpoint of the edges adjacent to V_i takes $O(ck)$ cost. Finding the new endpoint of an edge in category 3 requires constant cost after an $O(n/k)$ preprocessing of the Euler tour of the cluster spanning tree. The number of neighbor clusters is $O(k)$ so checking the cluster labels and adding edges costs no more than $O(k)$. The overall cost to construct one local graph is thus $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*. After plugging in c is $O(k)$ in expectation and $O(k \log n)$ *whp*, the overall cost matches the bounds in the lemma. ◀

Queries.

With the local graph and the BC labeling on the clusters graph, all sorts of biconnectivity queries can be made. Some of them are easier while other queries require more steps, and the preprocess steps are shown in an overview of 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 22.** *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 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 $O(k^2)$ on average and $O(k^2 \log n)$ *whp*.

Articulation points. By a similar argument that a vertex is an articulation point of the original graph if and only if it is an articulation point of the associated local graph, checking whether articulation points in a cluster costs $O(k^2)$ on average and $O(k^2 \log n)$ *whp*.

We now discuss how some more complex queries can be made. To start with, we show some definitions and results that are used in the algorithms for queries.

▶ **Definition 23** (root biconnectivity)**.** 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 indicate 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 cluster C that disconnects v from the outside vertex of the cluster root.

▶ **Lemma 24.** *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 straight-forward. The cost to construct the local graphs and compute root biconnectivity is $O(nk)$, and since 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-style 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 with constant cost) 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 cost to check whether any articulation point disconnects the paths in C_1, C_2 and C_{LCA}. For the rest of the clusters, since we have already precomputed and stored the root biconnectivity of all outside vertices, then applying a leafix on the clusters spanning tree computes the cluster containing the articulation point of each cluster root. Therefore checking whether such an articulation point on the path between C_1 and C_{LCA} or between C_2 and C_{LCA} that disconnects v_1 and v_2 takes $O(1)$ cost. Therefore checking whether two vertices are biconnected requires $O(k^2)$ cost in expectation and no writes.

Querying whether two vertices are 1-edge connected. This is a similar query comparing to the previous one and the only difference is whether an edge, instead of a vertex, is

able to disconnect two vertices. The query can be answered in a similar way by checking whether a bridge disconnects the two vertices on their spanning tree path, which is related to the two clusters containing the two query vertices and the LCA cluster, and the precomputed information for the clusters on the tree path among these three clusters. The cost for a query is also $O(k^2)$ operations in expectation and it requires no writes.

Queries on biconnected-component labels for edges. We now answer the standard queries [12, 25] of biconnected components: given an edge, report a unique label that represents the biconnected component this edge belongs to.

We have already described the algorithm to check whether any two vertices are biconnected, so the next step is to assign a unique label of each biconnected components, which requires the following lemma:

▶ **Lemma 25.** *A vertex in one cluster is either in a biconnected component that only contains vertices in this cluster, or biconnected with at least one outside vertex of this cluster.*

Proof. Assume a vertex v_1 in this cluster C is biconnected to another vertex v_2 outside the cluster, then let v_o be the outside vertex of C on the spanning tree path between v_1 and v_2, and v_1 is biconnected with v_o, which proves the lemma. ◀

With this lemma, we first compute and store the labels of the biconnected components on the cluster roots, which can be finished with $O(nk)$ expected operations and $O(n/k)$ writes with the BC labeling on the clusters graph and the the root biconnectivity of outside vertices on all clusters. Then for each cluster we count the number of biconnected components completely within this cluster. Finally we apply a prefix sum on the numbers for the clusters to provide a unique label of each biconnected component in every cluster. Although not explicitly stored, the vertex labels in each cluster can be regenerated with $O(k^2)$ operations in expectation and $O(k^2 \log n)$ operations *whp*, and a vertex label is either the same as that of an outside vertex which is precomputed, or a relative label within the cluster plus the offset of this cluster.

Similar to the algorithm discussed in Section G.2, when a query comes, the edge can either be a cluster tree edge, a cross edge, or within a cluster. For the first case the label biconnected component is the precomputed label for the (lower) cluster root. For the second case we just report the vertex label of an arbitrary endpoint, and similarly for the third case the output is the vertex label of the lower vertex in the cluster. The cost of a query is $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*.

With the concepts and lemmas in this section, with a precomputation of $O(nk)$ cost and $O(n/k)$ writes, we can also do a normal query with $O(k^2)$ cost in expectation and $O(k^2 \log n)$ *whp* on **bridge-block tree**, **cut-block tree**, and **1-edge-connected components**.

G.4 Parallelizing Biconnectivity Algorithms

The two biconnectivity algorithms discussed in this section are essentially highly parallelizable. The key algorithmic components include Euler-tour construction, tree contraction, graph connectivity, prefix sum, and preprocessing LCA queries on the spanning tree. Since the algorithms run each of the components a constant number of times, and the depth of the

algorithm is bounded by the depth of graph connectivity, whose bound is provided in Section 4 ($O(\omega^2 \log^2 n)$ and $O(\omega^{3/2} \log^3 n)$ *whp* respectively when plugging in β as $1/\omega$ and $1/\sqrt{\omega}$).[6]

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 any computations within a cluster. This term is additive to the overall depth, since after acquiring 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 ones can be queried in parallel.

H Sublinear-Write Algorithms on Unbounded-Degree Graphs

Here we discuss a solution to generate another graph G' which has bounded degree with $O(m)$ vertices and edges, and the connectivity queries on the original graph G can be answered in G' equivalently.

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. Considering a star with all other vertices connecting to a specific vertex v_1, we build a binary tree structure with 2 virtual nodes on the first level $v_{1,2\to n/2}$, $v_{1,n/2+1\to n}$, 4 virtual nodes on the second level $v_{1,2\to n/4}, \cdots, v_{1,3n/4+1\to n}$ 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.

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 $O(m)$ 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.

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