Log(Graph): A Near-Optimal High-Performance Graph Representation

ABSTRACT
Various graphs such as web or social networks may contain up to hundreds of billions of edges, yet they are not necessarily stored and thus managed efficiently. To address this, we propose Log(Graph): a graph representation that enables cheaper and faster graph processing for various families of graphs. The core idea is to encode the graph so that various elements approach or match respective storage lower bounds. We call our approach “graph logarithmization” because logarithmic terms are applied to various graph elements to derive the respective bounds. We use various techniques and notions to achieve this, including succinctness, compactness, and integer linear programming. We also show a high-performance design of Log(Graph) that uses modern bitwise operations, performance models, and parallel implementation for higher speedups. Our evaluation illustrates that Log(Graph) achieves simplicity and tunable performance as well as compression ratios. Log(Graph) can be used to improve the design of any graph processing framework, algorithm implementation, or library on both fat shared-memory NUMA nodes and distributed-memory data centers and supercomputers.

1 INTRODUCTION
Large graphs are behind many problems in machine learning, medicine, social network analysis, and computational sciences [54]. Lowering the size of such graphs (e.g., through compression) is becoming increasingly important for HPC and Big Data. First, it reduces expensive I/O. Next, it potentially improves performance by storing a larger fraction of data in caches. Third, it decreases the amount of hardware resources required to store a given graph.

Traditional lossless graph compression systems such as the well-known WebGraph [12] introduce costly decompression. This is often caused by the underlying complexity (e.g., WebGraph combines various techniques, such as sophisticated variable-length codes, reference encoding, and storing intervals instead of vertex IDs). Thus, one needs both simplicity and high performance for lossless compression.

In this work, we develop Log(Graph): a graph representation that achieves the above goals. Log(Graph) is founded on two fundamental ideas. First, to enable significant storage reductions, it applies storage lower bounds to various elements of the popular adjacency array graph representation. We call this approach “logarithmization” because the well-known combinatorial argument states that a lower bound to store an object from some set (or class) is the logarithm of the cardinality of this set (or class); the name Log(Graph) indicates that we “logarithmize” various elements of a graph representation such as vertex IDs or adjacency offsets. Second, we ensure that all the “logarithmization” schemes maintain high performance. The result is a representation that combines storage reductions with fast accesses to the graph data.

To motivate and explain the “logarithmization” idea for vertex IDs, consider using a fixed-size 64- and 32-bit memory word to respectively store a vertex ID and an edge weight (a strategy used in miscellaneous graph processing codes such as the GAP Benchmark Suite [8]). Yet, we note that the storage lower bound (for a graph with \( n \) vertices and the maximum edge weight \( W_{\text{max}} \)) associated with vertex IDs and edge weights is \( \log n \) and \( \log W_{\text{max}} \) bits. Now, for a graph with \( n = 2^{37} \) and \( W_{\text{max}} = 2^{16} \), 42% of each word with a vertex ID and 69% of each word with an edge weight is wasted. We instead “logarithmize” IDs and use \( \log n \) bits to store an ID and \( \log W_{\text{max}} \) bits to store an edge weight.

Other elements of a graph representation use more sophisticated designs. For example, we “logarithmize” an offset array by implementing it with a bit vector and then using an interesting family of succinct designs [45, 64] that approach theoretical storage lower bounds while enabling constant-time queries, without compressing or decompressing the bit vector data. We show that they can provide storage-efficient offsets into the adjacency data, asymptotically reducing the usual \( O(n \log n) \) bits used in traditional offset arrays.

Figure 1: (§ 1, § 7.2) The performance of Log(Graph) with the Single Source Shortest Path algorithm when logarithmizing vertex IDs.

Now, these ideas offer simplicity but come with challenges to be solved and questions to be answered to achieve high performance. For example, if we use \( \log n \) bits for a vertex ID instead of a fixed-size memory word, how do we ensure fast extraction of the graph connectivity information? Can we use fast bitwise operations available in today’s architectures? Which ones? How to logarithmize other graph elements?

In this work, we answer these and other questions and propose a set of schemes that are portable and straightforward to implement in any graph processing code, framework, or library. We motivate Log(Graph) in Figure 1 that illustrates example results for the Single Source Shortest Path (SSSP) algorithm running over traditional adjacency arrays and a selected Log(Graph) variant. Log(Graph) does not only reduce storage requirements, but also reduces data transfers, accelerating SSSP. The specific contributions are as follows:
We evaluate Log(Graph) on various graphs (synthetic and real-world) with a broad set of algorithms (BFS, PageRank, Connected Components, Betweenness Centrality, Triangle Counting, Single Source Shortest Paths). We show that graphs compressed with Log(Graph) can be processed as fast as uncompressed graphs with tuned graph processing codes. We also illustrate significant speedups over state-of-the-art compression systems such as WebGraph [12].

Finally, our work presents the first analysis of succinct data structures in a parallel setting, which is of independent interest.

## Representation Model

We present a unified model for graph representations. A representation is modeled as a tuple \((G, \Theta, \sigma, C[\Theta], C[\sigma])\). \(G\) is an offset structure that keeps the location of the adjacency structure \(\sigma\), of each vertex \(v\). \(\Theta\) and \(\sigma\) may be arbitrary data structures. \(C[\Theta]\) and \(C[\sigma]\) are compression schemes for \(\Theta\) and \(\sigma\). The size of the representation is \(|\Theta| + |\sigma|\) [bits] \((|\Theta|\) and \(|\sigma|\) denote the sizes of \(\Theta\) and \(\sigma\)).

**Machine and System Model** For more storage reductions on today’s hardware, we cover arbitrary hierarchical machines where, for example, cores reside on a socket, sockets constitute a node, and nodes form a rack. \(N\) is the number of hierarchy levels and \(H_i\) is the total number of elements from level \(i\). The first level corresponds to the whole machine; thus \(H_1 = 1\). We also refer specifically to the number of compute nodes as \(N_{\text{nodes}}\). Finally, the numbers of used threads per node and processes are \(T\) and \(P\). The memory word size is \(W\).

## 2.2 Used Concepts

We next explain concepts related to the structures used in Log(Graph) and their size; see also Figure 2 for an overview.

**Succinctness** Assume \(OPT\) is the optimal number of bits to store some data. A representation of this data is compact if it uses \(O(OPT)\) bits, succinct if it uses \(OPT + o(OPT)\) bits, and implicit if it takes \(OPT + O(1)\) bits [29]. They all should support a reasonable set of queries in (ideally) \(O(1)\) time [10].

**Compression** Traditional compression mechanisms such as zlib [31] differ from succinct schemes as the latter can be accessed without expensive decompression.

**Condensing** To avoid confusion, we use the term condensing to refer in general to reducing the size of some data.

We also describe the concept of graph separability. Intuitively, \(G\) is vertex (or edge) separable (i.e., has good separators) if we can divide \(V\) into two subsets of vertices of approximately the same size so that the size of a vertex (or edge) cut between these two subsets is much smaller than \(|V|\).

## 2.3 Used Data Structures

Finally, we describe structures used in Log(Graph): adjacency array that is the Log(Graph) foundation and succinct as well as compact bit vectors used to condense \(G\).

**Graph Adjacency Arrays** Log(Graph) builds upon the traditional adjacency array (AA) representation. An AA consists of a contiguous array with the adjacency data (\(\sigma\)) and an array with offsets to the neighbors of each vertex \(v\). The neighbors of each \(v\) form an array \(\sigma_v\) sorted by vertex IDs; all such arrays form a contiguous array \(\sigma\). \(\sigma_v\) is the offset to \(v\)'s adjacency data. An example AA is in Figure 2 (a).

**Bit Vectors and Rank/Select Queries** The next data structure are simple bit vectors that we will use to enhance \(G\) (§ 4.1, § 4.3). A bit vector \(S\) of length \(L\) takes only \(L\) bits, but uses \(O(L)\) time to answer two important queries: \(rank_5(x)\) and \(select_5(x)\). For a given \(S\), \(rank_5(x)\) returns the number of ones in \(S\) up to and including the \(x\)th bit. Conversely, \(select_5(x)\) returns the position of the \(x\)th one in \(S\). Now, many designs with \(rank\) and \(select\) answering in \(O(L)\) time have been proposed. An important family are succinct bit vectors.

**Succinct Bit Vectors** Succinct bit vectors of length \(L\) use \(L + o(L)\) bits. The first term is the amount of space for storing the actual data. The second term is the space for an additional structure that enables answering \(rank/\)select in...
Logarithmize other elements (§3.3 - §3.4)

3.2 Logarithmization of Vertex IDs

We first consider vertex IDs.

3.2.1 Vertex IDs (The Global Approach). The first and simplest step in Log(Graph) is to use the storage lower bound of a vertex ID in a graph with \( n \) vertices. Assume that a vertex ID uses \( W \) bits of space. In various AA designs, \( W \in \{32, 64\} \) bits and it thus corresponds to the size of 32- or 64-bit memory word. Log(Graph) extends this approach and uses the lowest applicable value: \( W = \lceil \log n \rceil \) bits, giving

\[
|\alpha| = 2m\lceil\log n\rceil
\]

3.2.2 Vertex IDs: The Local Approach. The advantage of the global approach described above lies in simplicity. Yet, even if it uses an optimal number or bits to store \( n \) vertex IDs, it may be far from optimal when considering subsets of these vertices. For example, consider a vertex \( v \) such that \( d_v < n \) and \( N_v < n \). Here, using \( \lceil \log n \rceil \) bits for a vertex ID in \( v \)'s adjacency list results in unnecessary storage overheads as one may need much fewer then \( \lceil \log n \rceil \) bits for a short adjacency list. Here, we use \( \lceil \log N_v \rceil \) bits to store a vertex ID in \( \alpha \). The tradeoff is that one must also keep the information on the number of bits required for each \( v \). We use the fixed number of bits; it is lower bounded by \( \lceil \log \log N_v \rceil \) bits. Then

\[
|\alpha| = \sum_{v = 1}^{n} \left( d_v \lceil \log N_v \rceil + \lceil \log \log N_v \rceil \right)
\]
with ILP and gap encoding (for less storage) and efficient design (for more performance).

3.6 Incorporating Integer Linear Programming

Here, we reduce the size of vertex IDs by augmenting local logarithmization (§ 3.2). Consider any $\mathcal{A}_0$. We observe that a single neighbor in $\mathcal{A}_0$ with a large ID may vastly increase $|\mathcal{A}'|$: we use $\lfloor \log \tilde{N}_v \rfloor$ bits to store each neighbor in $\mathcal{A}_0$ but other neighbors may have much lower IDs. Thus, we reorder vertex IDs to preserve the graph structure but reduce maximal IDs in as many neighborhoods as possible. We illustrate an ILP formulation and then propose a heuristic. The new objective function is presented in Eq. (15). It minimizes the weighted sum of $\tilde{N}_v$ for all $v \in V$ (Eq. 15). Each maximal ID is given a positive weight which is the inverse of the neighborhood size ($d_v$); this intuitively decreases $\tilde{N}_v$ in smaller $\mathcal{A}_0$.

$$ \min \sum_{v \in V} \tilde{N}_v \cdot \frac{1}{d_v} $$

In Constraint (11), we set $\tilde{N}_v$ to be the maximum of the new IDs assigned to the neighbors of $v$.

Listing 1 describes a greedy polynomial-time heuristic for changing IDs. We sort vertices in non-decreasing order of their degrees (Line 5). Next, we traverse vertices in the sorted order, beginning with the smallest $|\mathcal{A}_0|$ and assign a new smallest ID possible (Line 6). The remaining vertices that are not a part of any $N_v$ are relabeled in Line 10. This scheme acts similarly to the proposed ILP.

1 // Input: G, Output: a new relabeling $\mathcal{P}(v), \forall v \in V$. */
2 void relabel(G) {
3   ID[0:n-1] = [0:n-1]; D[0:n-1] = [0-D[n-1]];
4   visit[0:n-1] = [false false]; ml = 1;
5   sort(ID); sort(D);
6   for(int i = 1; i < n; ++i) {
7       for(int j = 0; j < D[i]; ++j) {  
8           if(visit[i] == false) { $\mathcal{P}(i) = ml$; visit[i] = true; ; }
9           if(visit[i] == false) { $\mathcal{P}(i) = ml$; visit[i] = true; ; }
10       }
11     }
12   }

Listing 1: (§ 3.6) The greedy heuristics for vertex relabeling.

3.7 Incorporating Fixed-Size Gap Encoding

We next use the popular gap encoding technique to further reduce $|\mathcal{A}'|$. Traditionally in gap encoding one calculates the difference between all consecutive neighbors in each $\mathcal{A}_0$. As it may be arbitrary, it is then encoded with a variable-length code such as Varint [19]. We observe that this may entail significant decoding overheads. To alleviate this, we use fixed-size gap encoding where the maximum difference within a given
domain determines the number of bits used to encode this and other differences. In the global approach, the maximum difference for the whole \( \mathcal{A} \) is used to calculate the number of bits used to encode any other difference. In the local approach, the maximum difference in each \( \mathcal{A}_e \) determines the number of bits to encode any difference in the same \( \mathcal{A}_e \).

### 3.8 High-Performance Implementation

We finally describe the high-performance implementation. We focus on the global approach due to space constraints, the local approach entails an almost identical design and is presented fully in the technical report\(^1\).

**Bitwise Operations** We analyzed Intel bitwise operations to ensure the fastest implementation. Table 3 presents the used operations together with the number of CPU cycles that each operation requires [39].

<table>
<thead>
<tr>
<th>Name</th>
<th>C++ syntax</th>
<th>Description</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEXTR</td>
<td>_bextr_u64</td>
<td>Extracts a contiguous number of bits.</td>
<td>2</td>
</tr>
<tr>
<td>SHR</td>
<td>&gt;&gt;</td>
<td>Shifts the bits in the value to the right.</td>
<td>1</td>
</tr>
<tr>
<td>AND &amp;</td>
<td>&amp;</td>
<td>Performs a bitwise AND operation.</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>+</td>
<td>Performs an addition between two values.</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: (§ 3.8) The utilized Intel bitwise operations.

### Accessing an Edge (\( N_i \))

We first describe how to access a given edge in \( \mathcal{A} \) that corresponds to a given neighbor of \( v \) (\( N_i \)), see Listing 2 for details. The main issue is to access an \( s \)-bit value from a byte-addressable memory with \( s = \lceil \log n \rceil \).

In short, we fetch a 64-bit word that contains the required \( s \)-bit edge. In more detail, we first load the offset \( \theta[v] \) of \( v \)'s neighbors' array (Line 3). This usually involves a cache miss, taking \( T_{cm} \). Second, we derive \( o = s \cdot \theta[v] \) (the exact bit position of \( N_i \)); it takes \( T_{mul} \) (Line 3). Third, we find the closest byte alignment before \( o \) by right-shifting \( o \) by 3 bits, taking \( T_{bxr} \) (Line 4). Instead of byte alignment we also considered any other alignment but it entailed negligible (<1%) performance differences. Next, we derive the distance \( d \) from this alignment with a bitwise and acting on \( o \) and binary 111, taking \( T_{and} \). We can then access the derived 64-bit value; this involves another cache miss (\( T_{cm} \)). If we shift this value by \( d \) bits and mask it, we obtain \( N_i(v) \). Here, we use the x86 _bextr_ instruction that combines these two operations and takes \( T_{bxr} \). In the local approach, we also maintain the bit length for each neighborhood. It is stored next to the associated offset to avoid another cache miss.

### Accessing an Edge Weight

We store the weight of each edge directly after the corresponding vertex ID in \( \mathcal{A} \). The downside is that every weight is thus stored twice for undirected graphs. However, it enables accessing the weight and the vertex ID together. We model fetching the ID and the weight as a single cache line miss overhead \( T_{cm} \).

**Performance Model** We finally present the performance model that we use to understand better the behavior of \( \text{Log(Graph)} \). First, we model accessing an edge \( (N_i, \nu) \) as

\[
T_{edge} = 2T_{cm} + T_{mul} + T_{add} + T_{bxr}
\]

The model for accessing \( N_i \) looks similar with the difference that only one multiplication is used:

\[
T_{neigh}(v) = T_{edge} + (d_v - 1)(T_{add} + T_{bxr} + T_{and})
\]

As \( \mathcal{A} \) is contiguous we assume there are no more cache misses from prefetching. Now, we model the latency of \( d_v \) as

\[
T_{degree} = 2T_{cm} + T_{sub}
\]

### 4 LOGARITHMIZING THE \( \mathcal{O} \) STRUCTURES

We now logarithmize the whole offset structure \( \mathcal{O} \), treating it as a single entity with its own associated storage lower bound; see Figure 2 (b). For this, we use bit vectors instead of offset arrays and then apply succinctness to approach the storage lower bound, significantly reducing space for some graphs.

#### 4.1 Offset Arrays vs. Bit Vectors

Usually, \( \mathcal{O} \) is an array of \( n \) offsets and \( |\mathcal{O}| \) is much smaller than \(|\mathcal{A}|\). Still, in sparse graphs with low maximal degree \( d \), \(|\mathcal{O}| \approx |\mathcal{A}| \) or even \(|\mathcal{O}| > |\mathcal{A}| \). For example, for the USA road network, if \( \mathcal{O} \) contains 32-bit offsets, \(|\mathcal{O}| \approx 0.85|\mathcal{A}| \).

To reduce \(|\mathcal{O}| \), one can use a bit vector. For this, \( \mathcal{A} \) is divided into blocks (e.g., bytes or words) of a size \( B \) [bits]. Then, if \( j \)-th bit of \( \mathcal{O} \) is set (i.e., \( \mathcal{O}[i] = 1 \)) and if this is \( j \)-th set bit in \( \mathcal{O} \), then \( \mathcal{A} \) starts at \( j \)-th block. The key insight is that getting the position of \( j \)-th set bit and thus the offset of vector \( j \) is equivalent to performing select\( \mathcal{O} \) (cf. § 2.3).

Yet, select on a raw bit vector takes \( O(n) \) time. Thus, we first incorporate two designs that enhance select: Plain (bvPL) and Interleaved (bvIL) bit vectors [36]. They both trade some space for faster select. bvPL uses up to 0.25\(|\mathcal{O}| \) additional bits in an auxiliary data structure to enable select in \( O(1) \) time. In bvIL, the original bit vector data is interleaved (every \( L \) bits) with 64-bit cumulative sums of set bits up to given positions; select has \( O(\log |\mathcal{O}|) \) time [36]. Neither bvPL nor bvIL achieve succinctness; we use them (1) as reference points and (2) because they also enable a smaller yet simple \( \mathcal{O} \).

#### 4.2 Understanding Storage Lower Bounds

A bit vector that serves as an \( \mathcal{O} \) and corresponds to an offset array of a \( G \) takes \( 2^{\log \theta} \) bits. This is because it must be able to address up to \( 2m \cdot W \) bits (there are \( 2m \) edges in \( \mathcal{A} \), each stored using \( W \) bits) grouped in blocks of size \( B \) bits. Now, there are exactly \( L = \left( \frac{2^{\log \theta}}{n} \right) \) bit vectors of length \( \frac{2^{\log \theta}}{B} \) with \( n \) ones and the storage lower bound is thus \([L] \) bits.

#### 4.3 Incorporating Succinctness

To condense \( \mathcal{O} \) further, we propose to use succinct bit vectors. Note that these are simply various forms of \( \mathcal{O} \) and they do not entail traditional compression. First, we use the entropy

---

1. To be released after the publication due to double-blindness.

---

**Listing 2 (§ 3.8) Accessing an edge in Log(Graph) (\( N_i(v) \)).**

```c
1 /* v_ID is an opaque type for IDs of vertices. */
2 v_ID N_i(v_ID v, int32_t i, int64_t* \theta, int64_t* \mathcal{A}, int8_t s){
3   int64_t exactBitOffset = s * *(\theta(v) + i);
4   int8_t* address = (int8_t*) \theta * (exactBitOffset >> 3);
5   int64_t distance = exactBitOffset & 7;
6   int64_t value = ((int64_t)*(address)[8]);
7   return _bextr_u64(value, distance, s); }
```

---
based bit vector (bvEN) [24, 66]. The key idea behind bvEN is to use a dictionary data structure [16] that achieves the lower bound for storing bit vectors of length $2Wm/B$ with $n$ ones that is presented in §4.2. Second, we use sparse succinct bit vectors (bvSD) [64]. Here, positions of ones are represented as a sequence of integers, which is then encoded using the Elias-Fano scheme for non-decreasing sequences. As bvSD specifically targets sparse bit vectors, we expect it to be a good match for various graphs where $m = O(n)$.

Third, we investigate the B-tree based bit vector (bvBT) [1]. This data structure supports inserts, making the bit vector dynamic. It is implemented with B-trees where leaves contain the actual bit vector data while internal nodes contain meta data for more performance. Finally, the gap-compressed (bvGC) dynamic variant is incorporated [1] along with bvSD also compresses sequences of zeros.

### 4.4 Theoretical Storage & Time Analysis

We now illustrate the storage/time complexity of the described offset structures in Table 4. For completeness, we present the asymptotic and the exact size as well the time to derive $\mathcal{E}_v$ but also rank and select queries. Now, ptrW together with bvPL and bvBT feature the fastest $\mathcal{E}_v$. Yet, we show in the evaluation (§7) that the hidden constant factors entail overheads for bit vectors. Simultaneously, their size outperforms that of ptrW for various sparse graphs.

### 4.5 High-Performance Implementation

For high performance, we use the sdsl-lite library [37] that provides fast codes of various succinct and compact bit vectors. Yet, it is fully sequential and oblivious to the utilized workload. Thus, we evaluate its performance tradeoffs (§7) and identify the best designs for respective graph applications and families, illustrating that the empirical results follow the theoretical analysis from §4.4.

### 5 LOGARITHMIZING THE $\mathcal{A}$ STRUCTURE

We now turn our attention to $\mathcal{A}$, the second part of AA; see Figure 2 (a). As $\mathcal{A}$ is more complex than $\mathcal{E}$, we first develop a formal model for logarithmizing $\mathcal{A}$ and show that various past schemes are merely its special cases. We illustrate that they all entail inherent performance or storage issues and we then design novel schemes to overcome these problems.

### 5.1 A Model for Logarithmizing $\mathcal{A}$

Log(Graph) comes with many logarithmization schemes $\mathcal{C}[\mathcal{A}]$ for condensing $\mathcal{A}$ that target various classes of graphs. To facilitate a unified reasoning about them, we can define any such logarithmization scheme to be a tuple $(\mathcal{P}, \mathcal{F})$, $\mathcal{P}$ is the permuter: a function that relabels the vertices. We introduce $\mathcal{P}$ to explicitly capture the notion that appropriate labeling of vertices significantly reduces $|\mathcal{A}|$. We have $\mathcal{P} : V \rightarrow N$ such that (the condition enforces the uniqueness of IDs):

$$\forall v,u \in V \quad (v \neq u) \Rightarrow [\mathcal{P}(v) \neq \mathcal{P}(u)]$$

Next, $\mathcal{F} = \{ \mathcal{F}_x \mid x \in \mathbb{N} \}$ is a set of transformers: functions that map sequences of vertex labels into sequences of bits:

$$\mathcal{F}_x : \mathcal{V} \times \ldots \times \mathcal{V} \rightarrow \{0, 1\} \times \ldots \times \{0, 1\}$$

We introduce $\mathcal{F}$ to enable arbitrary operations on sequences of relabeled vertices, for example be the Varint encoding [28].

### 5.2 Understanding Storage Lower Bounds

$\mathcal{A}$ is determined by the corresponding $G$ and thus a simple storage lower bound is determined by the number of graphs with $n$ vertices and $m$ edges and equals $\lceil \log \left( \frac{2^m}{m} \right) \rceil$ (Table 2).

Now, today’s graph codes already approach this bound. For example, the Graph500 benchmark [61] requires $\approx 1,126$ TB for a graph with $2^{22}$ vertices and $2^{46}$ edges while the corresponding lower bound is merely $\approx 350$ TB. We thus propose to assume more about $G$’s structure on top of the number of vertices and edges. We now target separable graphs (§2.2).

### 5.3 Incorporating Compactness

We use compact graph representations that take $O(n)$ bits to encode graphs. The main technique that ensures compactness that we incorporate is recursive bisectioning. We first describe an existing recursive bisectioning scheme (§5.3.1) and then enhance it for more performance (§5.3.2).

### 5.3.1 Recursive Bisectioning (RB)

Here, we first illustrate a representation introduced by Blandford et al. [10] (referred to as the RB scheme) that requires $O(n)$ bits to store a graph that is separable (see §2.2). Figure 4 contains an example. The basic method is to relabel vertices of a given graph $G$ to minimize differences between the labels of consecutive neighbors of each vertex $v$ in each adjacency list. Then, the differences are recorded with any variable-length gap encoding scheme such as Varint [28]. Assuming that the new labels of $v$’s neighbors do not differ significantly, the encoded gaps use less space than the IDs [10]. Now, to reassign labels in such a way that the storage is reduced, the graph (see 5 in Figure 4 for an example) is bisectioned recursively until the size of a partition is one vertex (for edge cuts) or a pair of connected vertices (for vertex cuts); in the example we focus on edge cuts. Respective partitions form a binary separator tree with the leaves being single vertices. Then, the vertices are relabeled as imposed by an inorder traversal over the leaves of the separator tree 5. The first leaf visited gets the lowest label (e.g., 0); the label being assigned is incremented for each new visited leaf. This minimizes the differences between the labels of the neighboring vertices (the leaves corresponding to the neighboring vertices are close to one another in the separator tree), reducing AA’s size 4, 5.
Using Permuters and Transformers One can easily express RB using \( \mathcal{P} \) and \( T \). First, \( \mathcal{P} \) relabels the vertices according to the order in which they appear as leaves in the inorder traversal of the separator tree obtained after recursive graph bipartitioning. Here, we partition graphs to make subgraphs [almost] equal \(<0.1%\) of difference in size. Second, each transformer \( T = \{ T(v, N_v) \} \) takes as input \( v \) and \( N_v \). It then encodes the differences between consecutive vertex labels using Varint. The respective differences are: \(|N_{1,v} - N_{0,v}|, N_{2,v} - N_{1,v}, \ldots \text{ and } N_{n,v} - N_{n-1,v}|.

Problems RB suffers from very expensive preprocessing, as we illustrate later in §7 (Table 6). Generation of RB usually takes more than 20x longer than that of AA.

5.3.2 Recursive Bisectioning (BBR). The core idea is to relabel vertices so that vertices in clusters have large common prefixes (clusters are identified during partitioning). One prefix is stored only once per each cluster.

What Does It Fix? BBR alleviates two issues inherent to RB. First, there is no costly inorder traversal over the separator tree. More importantly, there is no expensive derivation of the full separator tree. Instead, one sets the number of partitioning levels upfront to control the preprocessing overhead.

Permuter (Relabeling Vertices) We present an example in Figure 4. First, we recursively bipartition the input graph \( G \) to identify common prefixes and uniquely relabel the vertices. After the first partitioning, we label an arbitrarily selected prefix is stored only once per each cluster.

Transformer (Encoding Edges) Here, the idea is to group edges within each subgraph derived in the process of hierarchical vertex labeling. Several leading bits are identical in each label and are stripped off, decreasing \( |v^t| \). To make such a hierarchical adjacency list decodable, we store (for each \( v \)) such labels of \( v \)'s neighbors from the same subgraph contiguously in memory, together with the common associated prefix and the neighbor count.

5.4 Incorporating Integer Linear Programming We next logarithmize \( G \) with ILP to target generic graphs and not just the ones that are separable. We first illustrate a simple existing scheme that uses ILP for graph storage reductions (§5.4.1) and then accelerate it (§5.4.2).

5.4.1 Optimal Difference-Based (ODB). There are several variants of ILP-based schemes [32] where the objective function minimizes: the sum of differences between consecutive neighbors in adjacency lists (minimum gap arrangement (MGapA)), the sum of logarithms of differences from MGapA (minimum logarithmic gap arrangement (MLogGapA)), the sum of differences of each pair of neighbors (minimum linear arrangement (MLinA)), and the sum of logarithms of differences from MLinA (minimum logarithmic arrangement (MLogA)).

Using Permuters and Transformers Now, ODB’s \( T \) is identical to that of RB as it encodes ID differences while \( \mathcal{P} \) determines the relabeling obtained by solving a respective ILP problem. Consider a vector \( v = (\mathcal{P}(v_1), \ldots, \mathcal{P}(v_n))^T \) that models new vertex labels (where \( v_1, \ldots, v_n \in V \)). For example, the MGapA and MLogA objective functions are respectively

\[
\begin{align*}
\min_{\mathcal{P}(v) : v \in V} & \sum_{v \in V} \sum_{i=0}^{n-1} |\mathcal{P}(N_{i+1,v}) - \mathcal{P}(N_{i,v})| \quad (14) \\
\min_{\mathcal{P}(v) : v \in V} & \sum_{v \in V} \sum_{i=0}^{n-1} \log|\mathcal{P}(v) - \mathcal{P}(u)| \quad (15)
\end{align*}
\]

Both functions use the uniqueness Constraint (12).

Problems All of the above schemes except MLogGapA were proved to be NP-hard [20] and do not scale with \( n \). MLogGapA is still an open problem.

5.4.2 Positive Optimal Differences (POD). We now enhance the ODB MGapA (§5.4.1) by removing the absolute value \( | \cdot | \) from the objective function, which accelerates relabeling. Yet, this requires additional constraints to enforce that the neighbors of each vertex are sorted according to their IDs. We present the constraints below; readers who are not interested in the mathematical details may proceed to §5.5.

\[
\begin{align*}
\forall_{v \in V} \forall_{i,j \in [1..d_v]} & \left[ \mathcal{P}(N_{i,v}) + (x_{vij} - 1) \cdot n \leq N_{j,v} \right] \quad (16) \\
\forall_{v \in V} \forall_{i,j \in [1..d_v]} & \left[ \mathcal{P}(N_{i,v}) + (1 - x_{vij}) \cdot n \geq N_{j,v} \right] \quad (17) \\
\forall_{v \in V} \forall_{i,j \in [1..d_v]} & \left[ \sum_{i=1}^{d_v} x_{vij} = 1 \right] \quad (18) \\
\forall_{v \in V} \forall_{i,j \in [1..d_v]} & \left[ N_{i,v} < N_{i+1,v} \right] \quad (19)
\end{align*}
\]

\( N_{i,v} \) is the initial labeling of \( N_v \), \( x_{vij} \) is a boolean variable that determines if neighbor \( j \in N_{i,v} \) must be \( i \)th neighbor in \( N_v \) according to relabeling \( \mathcal{P} \). If \( x_{vij} = 1 \), constraints (16), (17) use \( N_{i,v} = \mathcal{P}(N_{i,v}) \), otherwise they are trivially satisfied. Constraint (18) selects each neighbor once. Finally, constraint (19) sorts \( N_v \) in the increasing label order.
5.5 Combining Compactness and ILP (CMB)
Finally, we design combining (CMB) schemes that use the compact recursive partitioning approach to enhance ODB and others. The core idea is to first bisect the graph k times (within the given time constraints), and then encode independently each subgraph (cluster) with a selected ILP scheme. We illustrate an example of this scheme in Figure 4.

What Does It Fix? First, the initial partitioning does not dominate the total runtime. Second, the NP-hardness of ODB is alleviated as it now runs on subgraphs that are k times smaller than the initial graph. Finally, it is generic and one can use an arbitrary scheme instead of ODB.

Using Permuters and Transformers The exact design of \( \mathcal{P} \) and \( \mathcal{T} \) depend on the scheme used for condensing subgraphs. For example, consider ODB. The most significant k bits are now determined by \( G \)'s partitioning. The remaining bits are derived from ODB independently for each subgraph. Their combination gives each final label. \( \mathcal{T} \) can be, e.g., Varint.

5.6 Incorporating Degree-Minimizing (DM)
The final step is to relabel vertices so that those with the highest degrees (and thus occurring more often in \( \mathcal{A} \)) receive the smallest labels. Then, in one scheme variant (DM, proposed in the past [3]), full labels are encoded using Varint ("f" stands for full). In another variant (DMd, offered in this work), labels are encoded as differences ("d" indicates differences), similarly to RB. Thus, |\( \mathcal{A} \)| is decreased as the edges that occur most often are stored using fewer bits.

What Does It Fix? First, DM trades some space reductions for faster accesses to \( \mathcal{A} \) compared to BRB (BRB's hierarchical encoding entails expensive queries). Second, it does not require costly recursive partitioning. Finally, we later (§ 7.4, § 7.5) show that DMd significantly outperforms DMf and matches the compression ratios of the WebGraph library [12].

Permuter/Transformer DM’s \( \mathcal{T} \) is identical to that of RB. DM’s \( \mathcal{P} \) differs as the relabeling is now purely guided by vertex degrees: higher \( d_v \) enforces lower \( \mathcal{A} \)'s label.

6 HIGH-PERFORMANCE LIBRARY
Past sections (§ 3–§ 5) illustrate a plethora of logarithmization schemes and enhancements for various graph families and scenarios. This large number poses design challenges. We now present the Log(Graph) C++ library that ensures: (1) a straightforward development, analysis, and comparison of graph representations composed of any of the proposed schemes, and (2) high-performance. Log(Graph) implements the provided model (§ 2.1, § 5.1). The code is available online2.

Extensibility We achieve this by implementing the model from § 2.1, § 5.1. Log(Graph) is divided into four modules (well-separated parts of code) that group variants of \( \mathcal{O} \), \( C[A] \), \( \mathcal{A} \), and \( C[\mathcal{A}] \). The \( C[\mathcal{A}] \) module further manages submodules for various types of \( \mathcal{P} \) and \( \mathcal{T} \). This enables us to seamlessly implement, analyze, and compare the described AA variants.

High Performance The combinations of the variants of \( \mathcal{O} \), \( C[A] \), \( \mathcal{A} \), \( \mathcal{P} \), and \( \mathcal{T} \) give many possible AA designs. For example, \( \mathcal{O} \) can be any succinct bit vector. Now, selecting a specific variant takes place in a performance-critical code part such as querying \( d \). We identify four C++ mechanisms for such selections: #if pragmas, virtual functions, runtime branches, and templates. The first one results in unmanageable complex code. The next two entail performance overheads. We thus use templates to reduce code complexity while retaining high performance. Listing 3 illustrates: the generic template class, the constructor of a representation, and a function for resolving \( N_e \). A new representation Graph only requires defining the offset structure (\( \mathcal{O} \)), the offset compression structure (\( C[A] \)), and the transformer (\( \mathcal{T} \) types).

Next, selecting a permutation can be done with simple branches based on the value of \( \mathcal{P} \).

Listing 3: (§ 6) A graph representation from the Log(Graph) library.

7 EVALUATION
We now illustrate the advantages of Log(Graph).

7.1 Evaluation Scope and Methodology
We first describe the evaluation scope and methodology.

Considered Algorithms We consider the following algorithms included in the GAP Benchmark Suite [8]: Breadth-First Search (BFS), PageRank (PR), Single Source Shortest Paths (SSSP), Betweenness Centrality (BC), Connected Components (CC), and Triangle Counting (TC). BFS, SSSP, and BC represent various types of traversals. PR is an iterative scheme where all the vertices are accessed in each iteration. CC represents protocols based on pointer-chasing. Finally, TC stands for non-iterative compute-intensive tasks.

• BFS: A state-of-the-art variant with direction-optimization and other enhancements that reduce data transfer [7, 8].
• SSSP: An optimized Δ-Stepping algorithm [8, 55, 59].
• CC: A variant of the Shiloach-Vishkin scheme [5, 67].
• BC: An enhanced Brandes’ scheme [8, 13, 56].
• PR: A variant without atomic operations [8].
• TC: An optimized algorithm that reduces the computational complexity by preprocessing the input graph [21].

Considered Graphs We analyze synthetic power-law (the Kronecker model [52]), synthetic uniform (the Erdős-Rényi model [33]), and real-world datasets (including SNAP [53], KONECT [50], DIMACS [30], and WebGraph [12]); see Table 5 for details. Now, for Kronecker graphs, we denote them with symbols \( s_k e \) where \( s \) is the scale (i.e., \( \log_2 n \)) and \( e \) is the average number of edges per vertex. Due to a large amount of data we present and discuss in detail a small subset; the remainder is in the Appendix or technical report2.

Experimental Setup and Architectures We use the following systems to cover various types of machines:

• CSCS Piz Daint is a Cray with various XC \( n \) nodes. Each XC50 compute node contains a 12-core HT-enabled Intel

---

2To be released after the publication due to double-blindness.
Xeon E5-2690 CPU with 64 GiB RAM. Each XC40 node contains two 18-core HT-enabled Intel Xeons E5-2695 CPUs with 64 GiB RAM. The interconnection is based on Cray’s Aries and it implements the Dragonfly topology [48]. The batch system is slurm 14.03.7. This machine represents massively parallel HPC machines.

- **Monte Leone** is an HP DL 360 Gen 9 system. One node has: two Intel E5-2667 v3 @ 3.2GHz Haswells (8 cores/socket), 2 hardware threads/core, 64 KB of L1 and 256 KB of L2 (per core), and 20 MB of L3 and 700 GB of RAM (per node). It represents machines with substantial amounts of memory.

**Evaluation Methodology** We use arithmetic mean for data summaries. We treat the first 1% of any performance data as warmup and we exclude it from the results. We gather enough data to compute the median and the nonparametric 95% confidence intervals.

### 7.2 Logarithmizing Fine Elements

We start with logarithmizing fine graph elements.

#### 7.2.1 Description of Preliminaries.

**Main Goal** We illustrate that logarithmizing fine graph elements, especially vertex IDs, reduces the size of graphs compared to the traditional adjacency arrays and incurs negligible performance overheads (in the worst case) or offers speedups (in the best case). We predict that the former is because of overheads from bitwise manipulations over the input data. Simultaneously, less pressure on the memory subsystem due to less data transferred to and from the CPU should result in performance improvements.

**Considered Log(Graph) Variants** We consider four variants of Log(Graph): LG-g (the global approach), LG-g-gap (the global approach with fixed-size gap encoding), LG-l (the local approach), and LG-l-gap (the local approach with fixed-size gap encoding). We also incorporate the ILP heuristics for relabeling from § 3.6 that enhances the local approach.

**Comparison Targets** We compare Log(Graph) to the traditional adjacency array (Trad). We do not aim to outperform sophisticated compression schemes but rather illustrate a straightforward logarithmization scheme that does not incur additional overheads but still ensures storage reductions and can be used in any graph processing library or engine.

#### 7.2.2 Key Analyses.

**Performance and Size** The results can be found in Figure 5. The collected data confirms our predictions. In many cases Log(Graph) offers performance comparable or better than that of the default adjacency array, for example for PR and SSSP. Simultaneously, it reduces $|\mathcal{V}|$ compared to Trad; the highest advantages are due to LG-l-gap ($\approx 35\%$ over LG-1); LG-g-gap does not improve much upon LG-g.

**Distributed-Memories and Scalability** The advantages from Log(Graph) directly extend to distributed-memories. Here,
we measure the amount of communicated data and compare it to Trad. For example, in a distributed BFS and for 1024 compute nodes, this amount is consistently reduced by 37% across all the studied graphs. We also conducted scalability analyses; the performance pattern is not surprising and all the Log(Graph) variants finish faster as $T$ increases. Full results can be found in the Appendix (§ 10.2.1, § 10.2.2).

7.2.3 Further Analyses. We also investigate the impact from ILP. It reduces the size of graphs and we obtain consistent improvements or 1-3%, for example from 0.614 GB to 0.604 GB for the ork graph. Yet, these improvements are significantly lower that those from gap-encoding.

7.2.4 Key Insights and Answers. The most important insights are as follows. First, logarithmizing fine elements does reduce storage for graphs while ensuring high-performance and scalability; both on shared- and distributed-memory machines. Second, both ILP (§ 3.6) and fixed-size (§ 3.7) gap encoding reduce $|\delta|$ with the latter being a definite winner.

7.3 Logarithmizing Offset Structures $\mathcal{O}$

We next proceed to analyze logarithmizing $\mathcal{O}$.

7.3.1 Description of Preliminaries.

Main Goal We illustrate that logarithmizing $\mathcal{O}$ with succinctness brings large storage reductions over simple bit vectors and offset arrays, without compromising performance.

Considered Log(Graph) Variants We investigate all the described $\mathcal{O}$ variants of offset arrays and bit vectors (§ 4.1, § 4.3). We also incorporate a variant where we logarithmize each offset treated as a fine-grained element, as described in § 3.4.

Investigated Parameters We vary the parameters related to the design of various types of $\mathcal{O}$: $L$, $B$, and $W$.

Comparison Targets We compare the above-mentioned $\mathcal{O}$ designs to the traditional zlib compression.

7.3.2 Key Analyses.

Size: Which Bit Vector is the Smallest? We first compare the size of all bit vectors for graphs of various sparsities $d$; see Figure 7. Static succinct bit vectors consistently use the least space. Interestingly, bvSD uses more space than bvEN for graphs with lower $d \leq 15$. This is because the term $\log \frac{\log n}{n}$ grows faster with the number of edges than that of bvEN.

Size: Bit Vectors or Offset Arrays? We next compare the size of offset arrays ($W \in \{32, 64, \lfloor \log n \rfloor \}) and selected bit vectors in Figure 16. As expected, offset arrays are the largest except for graphs with very high $d$. The sparser a graph, the bigger the advantage of bit vectors. Although $|\mathcal{O}|$ grows linearly with $m$ for bit vectors, the rate of growth is low (bit vector take only one bit for a single edge, assuming $B$ is one word size). Again, bvEN is the smallest in most cases.

Size: When To Condense $|\mathcal{O}|$? For many graphs, $|\mathcal{A}| \gg |\mathcal{O}|$ and condensing $|\mathcal{O}|$ brings only little improvement. Thus, we also investigate when to condense $|\mathcal{O}|$. We analyze the listed graphs and conclude that $|\mathcal{O}| \approx |\mathcal{A}|$ if the graph is sparse enough: $d \leq 5$. This is illustrated in Figure 8. In such cases, condensing $|\mathcal{O}|$ is at least as crucial as $|\mathcal{A}|$.

Size: Succinctness or Compression? We finally analyze the effect of traditional compression included in C(\mathcal{O}) on the example of the well-known zlib [31]. We present the results in Figure 9; (64-bit offset arrays represent all pointer schemes that followed similar results). Compressed variants of ptr64 and bvPL are ptr64C and bvPLC. Once more, the results heavily depend on $d$. For bit vectors, the sparser the graph, the more compressible $\mathcal{O}$ is; for $d < 10$, bvPL is smaller than both bvSD and bvEN. Interestingly, ptr64C is smaller than ptr64 by an identical ratio regardless of $d$. We conclude that zlib offers comparable or slightly (by up to $\approx 20\%$) better $|\mathcal{O}|$ reductions than succinct designs. We now proceed to show that these advantages are annihilated by performance penalties.

Performance Finally, we analyze the performance of various $\mathcal{O}$ designs queried by $T$ threads in parallel. Each thread fetches offsets of 1,000 random vertices. The results for twt and rca (representing graphs with high and low $d$) are in Figure 10. First, bvEN is consistently slowest due to its complex design; this dominates any advantages from its small size and better cache reuse. Surprisingly, bvEN is followed by bvL that has the biggest $|\mathcal{O}|$ (cf. Figure 16); its time/space tradeoff is thus not appealing for graph processing. Finally, bvPL and bvSD offer highest performance, with bvSD being the fastest for $T \leq 4$ (the difference becomes diluted for $T > 4$ due to more frequent cache line evictions). The results confirm the theory: bvPL offers $O(1)$ time accesses (while paying a high price in storage, cf. Figure 16) and bvSD uses little storage and fits well in cache. Next, we study offset arrays. ptr64 is the fastest for $T \leq 4$ due to least memory operations. Interestingly, the smaller $T$, the lower the latency of ptr64. We conjecture this is because fewer threads cause less traffic caused by the coherence protocol. As for zlib, it entails costly performance overheads as it requires decompression. We tried a modified blocked zlib variant without significant improvements.

7.3.3 Further Analyses. We vary the block size $B$ that controls the granularity of $\mathcal{A}$ to be an 8-bit byte or a 64-bit word; see Figure 8. First, larger $B$ reduces each $|\mathcal{O}|$ (as $|\mathcal{O}|$ is proportional to $B$). Next, $|\mathcal{A}|$ grows with $B$. This phenomenon is
similar to the internal fragmentation in memory allocation. Here, each \( \mathcal{A} \) is aligned with respect to \( B \). The larger \( B \), the more space may be wasted at the end of each array.

Other analyses are included in the Appendix (§ 10.3).

7.3.4 Key Insights and Answers. We conclude that succinct bit vectors are a good match for \( \mathcal{O} \). First, they reduce \( |\mathcal{O}| \) more than any offset array and are comparable to traditional compression methods such as zlib. Next, they closely match the performance of offset arrays for higher thread counts and are orders of magnitude faster than zlib. Finally, they consistently retain their advantages when varying the multitude of parameters, both related to input graphs (\( d \)) and to the utilized AA (\( B \) and \( \mathcal{A} \)). They can enhance any system for condensing static or slowly changing graphs that uses \( \mathcal{O} \).

7.4 Logarithmizing Adjacency Structures \( \mathcal{A} \)

We also evaluate the logarithmization of \( \mathcal{A} \).

7.4.1 Description of Preliminaries.

Main Goal We investigate the advantages and tradeoffs of the proposed schemes over the comparison targets.

Considered Log(Graph) Variants We evaluate the proposed schemes, including BRB (§ 5.3.2), POD (§ 5.4.2), and the combination of these two (§ 5.5); they are implemented using the proposed Log(Graph) library (§ 6).

Comparison Targets We consider all the described schemes expressed with permuters and transformers: RB (§ 5.3.1), ODB (§ 5.4.1), and DMd as well as DMf (§ 5.6). We also compare to the traditional adjacency array (Trad), and the state-of-the-art WebGraph (WG) [12] compression system.

7.4.2 Key Analyses.

BRB: Alleviating RB’s Preprocessing

We start with illustrating that BRB alleviates preprocessing overhead inherent to RB. Table 6 shows the overhead from RB compared to a simple AA. Now, BRB’s preprocessing takes equally long if we build the full separator tree. The idea is to build a given limited number of the separator tree levels. We illustrate this analysis in Figure 10. Using fewer partitioning levels increases \( |\mathcal{A}| \) but also reduces the preprocessing time (it approximately doubles for each new level). Interestingly, the storage overhead from preserving the recursive graph structure begins to dominate at a certain level, annihilating further \( |\mathcal{A}| \) reductions.

Yet, BRB comes with overheads while resolving \( N_v \) because one must construct vertex IDs from bit strings. This results in a 2-2.5x slowdown of obtaining \( N_v \), depending on the graph. We conclude that whether to use RB or BRB should depend on the targeted workload: for frequent accesses to \( N_v \) one should use RB while to handle large or evolving graphs that require continual preprocessing one should use BRB.

<table>
<thead>
<tr>
<th>Generation of RB [mb]</th>
<th>98.15</th>
<th>458.9</th>
<th>301.6</th>
<th>572.3</th>
<th>47.7</th>
<th>0.33</th>
<th>0.41</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation of AA [mb]</td>
<td>19.5</td>
<td>5.9</td>
<td>1.1</td>
<td>5.8</td>
<td>0.3</td>
<td>0.02</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 6: (§ 7.4.2) Illustration of preprocessing overheads of RB.

DMd: Approaching the Time/Space Sweetspot Next, we illustrate that DMd significantly reduces \( |\mathcal{A}| \), resolves \( N_v \), fast, outperforms WG, uses less storage than DMf, and can be generated fast. The size analysis is shown in Figure 11. We use relative sizes for clarity of presentation; the largest graphs use over 60 GB in size (in Trad). DMf and DMd generate much smaller \( \mathcal{A} \) than Trad, with DMd outperforming DMf, being comparable or in many cases better than either RB or BRB (e.g., for lj.m). Now, in various cases DMd closely matches WebGraph, for example for tw, fr, lj.m. For others, it gives slightly larger \( \mathcal{A} \) (e.g., for wik). Next, we also derive time to obtain \( N_v \); WG is consistently slower (>2x) than DMd; more results are in Figure 12 and the Appendix (§ 10.4.4). We conclude that DMd offers the storage/performance sweetspot: it ensures high level of condensing, trades a little storage for fast \( N_v \), and finally takes significantly (>10x for RB) less time to generate than any other scheme \( \mathcal{A} \).

7.4.3 Further Analyses. Other analyses include: investigating the ILP schemes, using various types of cuts while building the separator tree, and varying the maximum allowed difference in the sizes of subgraphs derived while partitioning. These analyses are included in the Appendix (§ 10.4). Here, we conclude that ILP does improve upon RB and DM by reducing sums of differences between consecutive IDs.

7.4.4 Key Insights and Answers. We conclude that BRB alleviates RB’s preprocessing overheads while DMd offers the best space/performance tradeoff.

7.5 The Log(Graph) Library

We finally evaluate the Log(Graph) library and show that it ensures high performance.

Performance: Graph Algorithms We use the Log(Graph) library to implement graph algorithms. We present the results for BFS, PR, and TC. We use succinct bit vectors (bvSD) as \( \mathcal{O} \) and various schemes for \( \mathcal{A} \). Our modular design based on the established model enables quick and easy implementation of \( \mathcal{A} \); each variant requires at most 20 lines of code. The results are shown in Figure 13. The BFS and PR analyses for large graphs (gho, orm, tw, usrn) illustrate that DMd is comparable to RB and DMf, merely up to 2x slower than the uncompressed Trad, and significantly faster (e.g., >3x for orm) than WebGraph. The relative differences for TC are smaller because the high computational complexity of TC makes decompression overheads less severe. Finally, we also study the differences between DMd and RB as well as DMf in more detail in Figure 13b) for a broader set of SNAP graphs. We conclude that DMd offers performance comparable to the state-of-the-art RB as well as DMf, while avoiding costly overheads from recursive partitioning.

Performance: Graph Accesses We also evaluate obtaining \( d_v \) and \( N_v \). This also enables understanding the performance of succinct structures in a parallel setting, which is of independent interest. Full results are in the Appendix (§ 10.4.4). Trad is the fastest (no decoding). The difference is especially visible for bvSD and \( f_c \) due to the complex \( \mathcal{O} \) design. DMd, DMf, and RB differ only marginally (1-3%) due to decoding.

8 RELATED WORK

We now discuss how Log(Graph) differs from or complements various aspects of graph processing.

8.1 Graph Frameworks, Standards, Languages

Log(Graph) aims to be a tool that enhances and complements any graph processing engine that stores graphs as adjacency arrays. Examples could be Pregel [57], Galois [49], GraphBLAS [58], CombBLAS [18], PBGL [40], GAPS [8], and Green-Marl [43]. For example, one could use logarithmized vertex IDs and use them within Galois to accelerate graph processing and reduce the pressure on the memory subsystem.
8.2 Reducing Size of Graphs

There exist numerous works on reducing the size of graphs.

Succinctness and Compactness Log(Graph) uses and improves succinct and compact designs to enhance graph storage and processing. A compact graph representation was proposed by Blandford et al. [10]. It reduces $|\mathcal{A}|$ for several real-world graphs. Yet, its preprocessing is costly. We alleviate this with our BRB scheme. Next, various succinct graph representations [6, 11, 47] are all theoretical structures with large hidden constants and no practical implementations or analyses. Third, there exist numerous succinct and compact bit vectors [29, 37, 38, 44, 45, 66, 70]. There are also designs that go below $n + o(n)$ bits with $o(n)$ time queries [24, 64]. Other schemes such as tree representations also exist [46, 60]. Finally, there are several libraries of succinct data structures [2, 27, 34–36, 41]. Contrarily to our work, none of these designs enhances graph processing and they do not address parallel settings.

Graph Compression A mature compression system for graphs is WebGraph [12]. It compresses $\mathcal{A}$ with: difference encoding, sharing parts of adjacency lists between vertices with similar neighborhoods, and storing consecutive vertex IDs as intervals. Besides WebGraph, there exists a large body of works on compressing webgraphs [3, 4, 14, 15, 17, 22, 23, 25, 42, 51, 62, 65, 68]. Some mention encoding some vertex IDs with the logarithmic number of bits; no details, analyses, or enhancements such as the local approach are provided [3]. Others collapse specified subgraphs into supernodes and merge edges between them into a superedge [17, 63, 65, 69]. Some of these works aim at generating a summary of input graphs for more efficient graph processing [63, 69]. Finally, several works use ILP to relabel vertices to reduce $|\mathcal{A}|$ [20, 32]. These systems often come with complex compression that requires costly decompression. Now, Log(Graph) embraces these schemes in its model and offers more flexibility. On one hand, it enables simple and generic logarithmization of fine elements that offers inexpensive storage reductions without or with negligible performance overheads. Simultaneously, it comes with more sophisticated schemes that can be used for graphs with more specific properties such as separability.

9 CONCLUSION

Reducing graph storage overheads is one of the key challenges for large-scale computations. Yet, as various studies show, traditional schemes such as zlib [31] or WebGraph [12] incur costly performance penalties.

In this work, we illustrate a graph representation called Log(Graph) that unifies various graph compression schemes and applies logarithmic storage lower bounds to (aka “logarithmize”): fine-grained graph elements such as vertex IDs, offset structures, and adjacency structures.

First, logarithmizing fine-grained elements offers simplicity and negligible performance overheads or even speedups from reducing the pressure on the memory subsystem and from incorporating the modern bitwise operations. It can be applied to any graph and can enhance any graph processing engine in shared- and distributed-memory settings. For example, we accelerate SSSP in the GAP Benchmark [8] by $\approx 20\%$ while reducing the required storage by 20-35%.

Second, for logarithmizing offset or adjacency data we incorporate more sophisticated succinct and compact data structures and techniques such as ILP. We investigate the associated tradeoffs and identify as well as tackle the related issues, enhancing the processing and storing of both specific and general graphs. For example, Log(Graph) outperforms WebGraph (>3x for the orr graph) while nearly matching its compression ratio with various schemes. We provide a careful and extensible implementation that ensures high performance and will be available online upon publication.

Finally, to the best of our knowledge, our work is the first performance analysis of succinct data structures in a parallel environment. It illustrates surprising differences between succinct bit vectors and offset arrays. We believe that our insights can be used by both theoreticians and practitioners to develop more efficient succinct schemes that would offer higher speedups in parallel settings.

REFERENCES

We now analyze in more detail how logarithmizing fine-grained graph elements impacts aspects such as scalability or communicated data.

10.2 Investigating Scalability. We also provide the results of scalability analyses. We vary the number of threads $T$ for various real-world graphs, see in Figure 14.

10.2.2 Investigating Distributed-Memory Settings. Finally, we also present results that show how Log(Graph) reduces the amount of communicated data in a distributed-memory environment when logarithmizing fine-grained graph elements. The results are illustrated in Figure 15.

10.3 Logarithmizing $\phi$: Additional Analyses

In the main body of the work, we have only analyzed the influence of $\phi$’s properties on $|\phi|$. Yet, the scope of the interplay between AA’s parameters is much broader: $|\phi|$ is also impacted by $\phi$’s design. Specifically, if $\phi$ is encoded using a scheme that shrinks adjacency arrays, such as Blandford’s RB scheme, then the size of $\phi$ based on pointer arrays should remain the same (as $W$ is fixed) while bvPL and bvSD should shrink. Succinct designs are harder to predict due to more complex dependencies; for example bvSD’s length also gets smaller but as the ratio of ones to zeros gets higher, $|\phi|$ may as well increase; a similar argument applies to bvEN. We now analyze these effects by comparing the original $|\phi|$ to $|\phi|$ after relabeling of vertices according to the inorder traversal of the separator tree as performed in the Blandford’s scheme; see Figure 16. As expected, all offset arrays remain identical because they only depend on fixed parameters. Contrarily, all the bit vectors shrink (e.g., bvPL and bvSD are on average $\approx 25\%$ and $\approx 12\%$ smaller). This is because the relabeled $\phi$ uses less storage, requiring shorter bit vectors.

10.4 Logarithmizing $\phi$: Additional Analyses

We also illustrate more analyses related to logarithmizing $\phi$.

10.4.1 Using Vertex Cuts Instead of Edge Cuts. So far, we have only considered edge cuts (ECs) in the considered recursive partitioning schemes (RB and BRB). Yet, as explained in § 5.3.1, vertex cuts (VCs) can also be incorporated to enhance RB. They seem especially attractive as it can be proven that they are always smaller or equal than the corresponding ECs [71]. In our setting, this relationship is more complicated as we partition graphs recursively and the correspondence between ECs and VCs is lost. Thus, we first compute the total sums of sizes of ECs and VCs at various levels of respective separator trees, see Figure 17. We ensure that the respective partitions are of almost equal sizes. We did not find strong correlation between cut sizes and sparsities $d$. We group selected representative graphs into social networks (SNs), purchase networks (PNs), and road graphs (RNs). For most SNs, ECs start from numbers much larger than in the corresponding VCs, and then steadily decrease. This is due to vertices with very high degrees whose edges are cut early during recursive partitioning. Contrarily, VCs in SNs start from very small values and then grow. This is because it is easy to partition initial input SNs using VCs as they are rich in communities [72], but later on, as communities become harder to find, cut sizes grow. Then, ECs and VCs in PNs follow similar patterns; they increase together with levels of separator trees. This suggests that in both cases it becomes more difficult to find clusters after several initial partitioning rounds. Finally, ECs and VCs in RNs differ marginally because these graphs are almost planar.

We conclude that, in most cases, VCs are significantly smaller than ECs, being potentially a more appealing tool in reducing $|\phi|$ because less information crosses partitions and has to be encoded as large differences in RB. Yet, $\phi$ based on VCs introduces redundancy as now some vertices are present in graph separators as well as in graph partitions. It requires additional lookup structures with shadow pointers [10] for mapping between such vertex clones (i.e., shadow vertex trees [10]). We calculated the number of shadow pointers in such structures and the resulting storage overheads in $|\phi|$ based on VCs. All the graphs follow similar trends. For example, twt requires 3.53M additional pointers, giving 4.81MB for $|\phi|$ (VCs, ptrLogn), as opposed to 3.1MB (ECs). Thus, the additional complexity from shadow vertex trees removes advantages from smaller cuts, motivating us to focus on ECs.

10.4.2 Relocating Balancedness of Partitions. While bisecting $G$, we now let the maximum relative imbalance between the
partition sizes be at most $\mathcal{D}$. We first analyze how changing $\mathcal{D}$ influences sizes of ECs and VCs for each level of separator trees. We plot the findings for representative graphs in Figure 18. As expected, cuts become smaller with growing $\mathcal{D}$: more imbalance more often prevents partitioning clusters. Yet, these differences in sizes of both ECs and VCs are surprisingly small. For example, the difference between ECs for twt (level 1) is only around $\approx 2\%$; other cases follow similar
patterns. They do not impact the final \(|d’|\), resulting in minor (\(\approx 1\%\)) differences.

10.4.3 Approaching the Optimal Labeling with POD/CMB. We now use POD and HYB to approach optimal labeling and outperform RB and DM. We use IBM CPLEX [26] to solve the ILP problems formulated in § 5.4.1 and § 5.4.2. We use two graphs \(g_1\) and \(g_2\), both consisting of two communities with few (<0.2\(n\)) edges in-between. Here, we illustrate (Figure 20) that POD/HYB do improve upon RB and DM by reducing sums of differences between consecutive labels. Each scheme is denoted as \(\mathcal{O}(y-z)\): \(y\) indicates the variant of the objective function (\(y=1\) for Eq. (14) and \(y=2\) for Eq. (15)) and \(z\) determines if we force the obtained labels to be contiguous (\(z=c\)) or not (\(z=u\)). Each proposed scheme finds a better labeling than RB (by 5-10\%) and DM (by 30-40\%).

10.4.4 Investigating Performance of Graph Accesses. Here, we present the full results of the performance of obtaining \(d_v\) and \(N_v\) that we discussed briefly in § 7.4.