New Methods for Graph Computation on Single Nodes

Zhixuan Zhou

Contents

1 Thesis Statement 3

2 Introduction 3

3 Related Work 6

3.1 Process Out-of-core Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
3.2 Process In-memory Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.3 Major Programming Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.4 Tension Between Parallelism and Iteration Count . . . . . . . . . . . . . . . . 8
3.5 Approximate Graph Computation . . . . . . . . . . . . . . . . . . . . . . . . 9

4 GraphZ: Degree-ordered Storage and Ordered Dynamic Messages 9

4.1 Degree-ordered Storage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
4.1.1 The Vertex Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
4.1.2 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
4.1.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
4.1.4 Analysis of Unique Degrees . . . . . . . . . . . . . . . . . . . . . . . . 12
4.1.5 Analysis of Edge Density Distribution . . . . . . . . . . . . . . . . . . . 13
4.2 Ordered Dynamic Messages . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

4.3 Empirical Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.3.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.3.2 Preprocessing And Vertex Index Size . . . . . . . . . . . . . . . . . . . . . 15
4.3.3 Extra Large Graph Performance . . . . . . . . . . . . . . . . . . . . . . 16
4.3.4 Large Graph Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.3.5 Medium Graph Performance . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3.6 Small Graph Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3.7 Performance Breakdown . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3.8 Bulk vs. Asynchronous Execution . . . . . . . . . . . . . . . . . . . . . 18
4.3.9 IO Statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5  **GraphStone: Hybrid Model and On the fly vertex assembly**  
5.1  Hybrid Execution Model  
5.2  Partitions in GraphStone  
5.3  On-the-fly Vertex Assembly  
5.4  Shuffling Graphs Without a Lookup Table  
5.5  Empirical Evaluation  
5.5.1  Experimental Setup  
5.5.2  Best-in-Class Performance  
5.5.3  In-memory Experimental Results  
5.5.4  Out-of-core Experimental Results  
5.5.5  Convergence Speed Study  
6  **Approximate in-memory compression and computation**  
6.1  Lossy Graph Compression  
6.2  Algorithms  
6.3  Preview Results  
7  **Future Plan**  
A  **Convergence in the HYBRID Model**  
A.1  Path Iteration Length  
A.2  Convergence Speed Analysis
1 Thesis Statement

This thesis presents new techniques in graph storage format, execution models and approximate computation for improving the performance of graph computation on single machines.

2 Introduction

Many important data sets are naturally structured as graphs, including the web and various social networks. Consequently, many frameworks have been proposed for analyzing and mining extremely large scale graphs. Frameworks support for graph analytics workloads requires delivering high performance—to return timely results to the analyst—while dealing with the large size of graphs. Until recently, the largest publicly available real graph was the YahooWeb graph which has 1.4B vertices and 6.6B edges [59]. The substantially larger Common Crawl Graph was recently released, which has 1.7B vertices and 64B edges [10]. While not public, Facebook reports internal graphs with over 1 trillion edges [7].

Given the importance of these workloads, there has been a recent explosion in specialized graph computing frameworks. This paper is concerned with two disjoint classes of framework that process graphs using a single machine:

1. **in-memory** systems designed with the assumption that main memory is large enough to hold the entire graph[36, 38, 45, 46] and

2. **out-of-core** systems that handle graphs larger than main memory by automating the tedious and error-prone process of transferring graph data between backing store and DRAM [18, 25, 31, 42, 65, 66].

In-memory frameworks are built for speed; e.g., Ligra achieves performance comparable to hand-tuned code [45], but cannot analyze graphs that exceed memory. Single-node out-of-core systems often out-perform distributed systems on very large graphs [25], but when run on small graphs that fit in memory they can be 3–5 \( \times \) slower than simple C programs [36].

Out-of-core frameworks are built for large graphs on small systems. In these cases, graphs must be processed out-of-core, meaning the disk becomes a fundamental bottleneck. GraphChi [25] supports out-of-core processing on a single machine while keeping the popular, easy-to-use **vertex-centric** programming model of distributed graph-processing systems like Pregel [35] and GraphLab [34].

This dichotomy between in-memory and out-of-core frameworks creates a dilemma for graph analysts: either use the out-of-core frameworks and suffer slow speed on smaller graphs or use two different frameworks—one for small graphs and another for large graphs.

Whether in-core or out-of-memory, a common programming model, known as vertex-centric (or gather-apply-scatter) [34], is shared by many graph analytics frameworks [1, 2, 15, 21, 25, 34, 35, 37, 43, 55]. In the vertex-centric model, users specify an update function and the graph engine repeatedly applies this update to each vertex. If an update affects an adjacent vertex, it can send a static message to that vertex. For example, in the GraphChi implementation of the PageRank algorithm, vertices update their rank, then send this new rank to adjacent vertices.
In subsequent updates, vertices read the messages and compute new ranks for themselves, again sending their new rank to all neighbors. The process repeats until the ranks converge.

Existing out-of-core graph analytics frameworks all rely on such static messages[18, 25, 42]. Between the time the message is sent and the time it is read, it must be stored. The storage required for messages is large—proportional to the number of edges [25, 42]. Reducing the storage requirement for messages would allow larger graphs to fit into the same memory, reducing IO pressure and greatly speeding up graph analytics.

A key design choice for the framework is synchronization between iterations. Bulk synchronous frameworks apply the update method using values from the previous iteration [9]. Asynchronous frameworks apply updates with the most recent data, even if it was produced during the same iteration [4]. Bulk synchronous approaches naturally parallelize and are used to maximize parallelism for in-memory approaches [45] and storage access locality in out-of-core systems [42]. Asynchronous approaches reduce the total number of iterations required for convergence, reducing total runtime for serial, or heavily IO-bound implementations [25].

There are two main challenges to delivering high-performance graph analytics both in-memory and out-of-core:

1. **Parallelism and Convergence Speed**: Both in-memory and out-of-core frameworks need parallelism for performance. Modern server-class systems require parallelism to achieve full memory and IO bandwidth utilization. As discussed above, however, bulk synchronous models sacrifice convergence speed for parallelism. For a random graph, asynchronous models converge nearly twice as fast as bulk synchronous models (see App. A) and empirical results are even better for natural graphs [25].

2. **Book-keeping Overhead**: When graphs are larger than memory, frameworks must track which vertices and edges are in memory and which are on disk. Additionally, messages to vertices that are on disk incur a high IO overhead as they are first written to disk and then reread when the destination vertex is later loaded. If a graph fits in memory, then the additional data structures and functions required for book-keeping only contribute overhead. If this book-keeping overhead is elided, then the framework simply cannot process graphs larger than the memory capacity.

Both challenges involve tradeoffs: (1) between parallelism and convergence speed, (2) between book-keeping overhead and the size of the graph the framework can handle and (3) between the accuracy and performance. Thus, these paper represents those techniques that delivers high performance for both in-memory and out-of-core graphs requires finding suitable compromises, useing the familiar gather-apply-scatter programming model, so it should be easy for users of existing frameworks to learn.

And according to the size of graphs and available memory, we classify those computation scenarios into 2 kinds: (1) **All out-of-core(AOOC)**: The memory is very limited, and the graph is very large, then the memory is too small to store all vertices in graphs and (2) **Edges out-of-core(EOOC)**: the memory is large enough to store all vertices but not all edges. AOOC often happens for desktop tier machines and EOOC is for workstation tier machines. We have 2 sets of techniques to help improve performance.
AOOC Methods: For AOOC, this thesis has following features that address the challenges listed above:

1. *degree-ordered storage* format that reduces memory requirements and disk accesses. The indices overhead would be super small and cost less time on preprocessing than GraphChi [25] and X-Stream [42]. Also, we could have less messages to be flushed to external storage thanks to power-law properties of natural graphs.

2. *ordered dynamic messages* helps apply messages directly in-memory. And because of the locality brought by *degree-ordered storage*, this would help reduce a lot of disk access.

We implement those techniques as GraphZ and compare the performance to GraphChi or X-Stream on SSD and HDD:

- When graphs fit in memory, there is no best system.

- For a graph that just exceeds memory, GraphZ provides a harmonic mean speedup 2.3-7.3× higher than GraphChi and 3.2-8.3× higher than X-Stream. For individual applications, GraphZ is up to 33× faster than GraphChi and 71× faster than X-Stream.

- For a graph that far exceeds memory, GraphZ’s harmonic mean speedups are 1.8-4.8× faster than GraphChi and 1.8-3× faster than X-Stream. Individual speedups are as high as 8× and 13× compared to GraphChi and X-Stream, respectively.

- For a graph that exceeds SSD capacity, GraphChi fails, but GraphZ’s harmonic mean speedup is 1.8× over X-Stream.

EOOC Methods: For EOOC, this thesis has following features that address the challenges listed above:

1. *A hybrid synchronous execution model*: The GraphStone runtime automatically divides the graph into partitions. Within partitions, updates are bulk synchronous, allowing the partition to be parallelized across multiple cores to maximize memory and IO utilization. Updates between partitions are asynchronous. For large graphs—with many partitions—most updates are asynchronous and iteration counts are close to those achieved by fully asynchronous models, but with much higher parallelism.

2. *On the fly vertex assembly*: All graph runtimes track vertices’ in- and out-edges. Internally, a vertex is thus the union of the user-defined vertex datatype and all the metadata associated with the vertex—referred to as a *full* vertex. Prior frameworks construct full vertices once, when the graph is first loaded. In contrast, GraphStone constructs full vertices on-the-fly, each time a partition is loaded. Since partitions are loaded multiple times, this process represents repeated computation, but the repeated computation allows GraphStone to save precious memory resources and fit many more vertices into memory at once.
3. User-defined message generation: The GraphStone framework requires users to specify a special function that generates a message. This user-defined function allows the GraphStone runtime to generate messages on the fly, much like it does full vertices, as described above. The combination of on-the-fly vertex generation and user-defined message generation greatly cuts down on the total disk IO required when the graph is processed out-of-core, without adding significant overhead when the graph is processed in-memory.

We implement above techniques as GraphStone in C++ for a Linux/x86 server and compare it to the best-in-class frameworks for both in-memory (Ligra [45]) and out-of-core processing (X-Stream [42]). We use a combination of publicly available natural graphs and simulated graphs. Overall we find:

- For graphs that fit in memory, GraphStone is $2.5 \times$ faster than Ligra. It is also over $7 \times$ faster than a hand-tuned serial C implementation and $36 \times$ faster than X-Stream, which is optimized for out-of-core.

- For out-of-core graph computations, GraphStone is $6.5 \times$ than X-Stream.

3 Related Work

3.1 Process Out-of-core Graphs

All graph analytics frameworks must account for the large sizes of natural graphs. For example, Facebook reports their internal graph data sets exceed 1 trillion edges [7].

There are 2 main approaches for processing large graphs that could not fit in memory: (1) Processing graphs in distributed systems and (2) Handling graphs on a single machine with external storage. Many graph frameworks deal with large data sets by distributing the graph over multiple physical machines’ memories. These include systems like Giraph [1], Pregel [35], GraphLab [34], PowerGraph [15], the Parallel Boost Graph Library [16], PowerLyra [6], and Hama [2]. A complete survey of distributed graph processing is beyond the scope of this thesis, which focuses on single machine graph analytics.

Somewhat surprisingly, such single-machine frameworks can match or even outperform distributed approaches [25, 42, 45]. We further divide the class of single-machine graph analytics frameworks into those that work when the graphs fit in main memory and those that can process graphs out-of-core.

GraphChi is the first system to support out-of-core graph processing, allowing large graphs to be analyzed on small machines [25]. GraphChi’s runtime automatically transfers data to and from backing the backing store. GraphChi uses asynchronous updates to reduce iteration count, achieving performance better than or equal to distributed systems.

GraphChi showed a single machine can efficiently process even large graphs, spawning number of additional out-of-core approaches in its wake. X-Stream outperforms GraphChi in many cases because it maximizes parallelism and disk read locality, but it requires migrating from the vertex-centric programming model to an edge-centric model [42]. TurboGraph [18] and FlashGraph [65] target high-end machines with arrays of fast SSDs. The edge-centric
model has been extended to further reduce IO by only loading needed edges, which reduces IO burden but not runtime [52]. GridGraph extends the edge-centric model using a two-level partitioning scheme [66]. While the edge-centric scheme improves performance, it requires the bulk synchronous processing model and tends to result in longer programs (see Table ??). Other approaches to IO reduction for out-of-core graph analytics include PathGraph [63], GraphQ [54], and MMap [31].

3.2 Process In-memory Graphs

In-memory approaches rely on machines with copious DRAM to process large graphs. One example is Ligra, which automatically alternates graph traversal patterns to achieve performance equivalent to hand optimized C code [45, 46]. Galois is an in-memory framework for all types of irregular processing, including graph analytics. Galois’s general approach to irregular processing can produce faster graph analysis in some cases [38]. Neither Ligra nor Galois support out-of-core processing, however, so users with graphs that exceed memory size must switch to another framework.

While those out-of-core approaches are fast when graphs exceed memory capacity, a recent study showed them to be several times slower than simple C code on graphs that fit in memory [36]. Thus prior work has created two disjoint approaches to single-machine graph processing: (1) use a fast framework (like Ligra or Galois) to handle small graphs, but rewrite in another framework (like GraphChi or X-Stream) if larger graphs need to be processed or (2) use an out-of-core framework, which handles large graphs but suffers slow performance for in-memory processing.

3.3 Major Programming Models

Exploiting the maximum parallelism is a key for improving performance. Natural graphs are generally unstructured, so it is hard to get data locality and dispatch tasks to workers equally. Vertex-centric is an method for splitting tasks among workers at a granularity of vertices, so that the computation could be load balanced among workers. Edge-centric is a more aggressive way to distribute tasks at a more fined granularity of edges, so that workload could get even better load balance. But edge-centric approaches often involve harder programming interfaces and only works for bulk synchronous (BSP).

Also, McSherry et al. question the need for any graph programming system [36]. They compare many of the above frameworks to simple PageRank [26] implementations written in standard programming languages without explicit graph support, finding that most frameworks produce substantially slower code and do not save lines of code (LOC) compared to the authors’ implementations.

We note two issues with this study. First, it conflates distributed computing approaches with those for out-of-core computing. Second, the study only compares performance for graphs that fit into memory on the test machine. This second issue is quite serious as the difficulty of analyzing graphs that do not fit into memory is the entire motivation for out-of-core approaches. And for in-memory graph Ligra [45], which focus on in-memory computation, is faster than naive C codes.
3.4 Tension Between Parallelism and Iteration Count

A key choice in graph framework design is the *execution model*. Two choices are apparent from the literature: bulk synchronous (BSP) [9] and asynchronous (ASYN) [4] execution. GraphStone proposes a hybrid model that combines properties of both bulk and asynchronous execution. We illustrate the difference between BSP and ASYN using breadth-first search (BFS) on the simple graphs in Fig. 1. We always iterate from left to right (from vertex 0 to 6), as all frameworks iterate through vertices in numerical order.

Table 1 compares the iteration counts for BSP and ASYN in the best and worst case. Table 1(a) shows the best case for ASYN. At iteration 0, vertex 0 is marked found, by both models. Then we go to iteration 1. For BSP, only vertex 1 can see its parent is found, so vertex 1 is found and marked. Vertices 2–6 see the same data as iteration 0, so they are not found in iteration 1. But for ASYN, vertex 1 sees its parent’s value is marked and is, itself, marked found. In the ASYN model, vertex 2 sees vertex 1’s latest value, so vertex 2 is marked found. The same process occurs with vertex 3–6, so all vertices are found during iteration 1 under ASYN. Next, we go to iteration 2. For BSP, only vertex 2 is found. For ASYN, since all vertices are found, there are no more iterations. In contrast, it takes BSP until iteration 6 to terminate.

Table 1(b) shows the worst case for ASYN. At iteration 0, vertex 6 is marked found in both models. Then we go to iteration 1. For BSP, only vertex 5 sees its parent is marked, so vertex 1 is found and marked. Vertices 0–4 see the same data as iteration 0, are not found in iteration 1. For ASYN, when we update 0–4, none of their parents are found. Only vertex 5 can see its parent is found. Iterations 2–6 are similar; in this case, ASYN requires the same iteration number as BSP.

In the best case, ASYN requires dramatically fewer iterations to converge. In the worst case, ASYN achieves the same convergence speed as BSP. On random graphs, ASYN is twice as fast as BSP (see Appendix A). On natural graphs, empirical results show it is much faster [25]. As mentioned above, BSP is much easier to parallelize, however.
Some prior work has addressed the tension between the parallelism of bulk synchronous processing and the convergence speed of asynchronous processing. ASPIRE gets good parallelism with a relaxed BSP model and constrains the staleness of vertices using a granularity of iterations [51]. Xie et al. propose a distributed system that switches between BSP and ASYN mode to adapt to network usage [58]. Haan and Daudjee break down the iteration barrier, allowing some vertices to be processed more often than others [17]. GiraphAsync uses asynchronous messages to adapt to network usage in distributed systems [32]. Wang et al. relax the BSP model to improve convergence speed [53]. These methods get faster convergence speed of BSP or improve parallelism of ASYN model, but none of them guarantee the convergence speed to be similar to ASYN.

GraphStone is the first unified framework that provides high-performance for both in-memory and out-of-memory graph analytics, and provides BSP-level parallelism with a convergence speed close to ASYN for random graphs. GraphStone exceeds the performance Ligra in-memory graphs and that of X-Stream on out-of-core processing. For a random graph, GraphStone has almost the same convergence speed as ASYN and same parallelism as BSP.

### 3.5 Approximate Graph Computation

Approximate results are often good enough for graph analytics. For algorithms like PageRank[26] and Randomwalk[33], we often only want the sorted top $k$ vertices. And users can often tolerate small errors in the results.

Shang’s results shows we could get the final results with less computation with small error[44]. KickStarter use approximate trimming and maintain the correctness for a set of algorithms[50]. There are also approaches working on the approximate graph computation from the algorithm side[8, 13, 40].

Graph sampling is another method for studying graph properties. There are a lot of sample methods, including Forest Fire, Random Walk, Vertices Sampling, Edge Sampling[19]. But certain sample method only works for certain graph algorithms[14, 20, 23, 27, 56, 64]. For some algorithms, the sampling process may even cost more than a running for the whole graph. We think we need a more general way to do the compression.

Above work shows that for many graph algorithms, approximate methods are practical. This thesis will talk about providing general graph compression methods and a general framework for major graph analytic algorithms and provide correctness studies.

### 4 GraphZ: Degree-ordered Storage and Ordered Dynamic Messages

#### 4.1 Degree-ordered Storage

While the number of vertices is small compared to the number of edges, efficient vertex access is crucial to performance. Degree-ordered storage (DOS) improves performance by compressing the vertex index a key data structure that tracks the location of each vertex in the file system – essential for out-of-core processing.
4.1.1 The Vertex Index

The vertex index allows the runtime to manage transfers between memory and disk. Many graph packages store the index using the compressed sparse rows (CSR) format [47], which requires an entry for every vertex. For large graphs, with many vertices, this index itself may be too large to fit in memory, meaning that vertex access requires one disk IO to load the appropriate part of the index and another to load the vertex itself.

In DOS, we sort vertices by decreasing out-degree, give each vertex a new ID based on this order, and update all the adjacency lists accordingly. Rather than store an index for every vertex, DOS simply stores the smallest id of those vertices with the same out-degree. The number of different out-degrees tends to be very small in natural graphs [15], so this format requires a (typically) much smaller number of indices than vertices, greatly reducing index size compared to CSR.

For the YahooWeb graph, the number of different degrees is less than 10k. DOS uses less than $16 \times 10,000 = 160$KB to hold the vertex index. In contrast, prefix-sum or CSR requires about $8 \times 1.4 \times 10^9 = 11.2$ GB. DOS’s tight format – almost four orders of magnitude reduction over common techniques – stores the entire index in memory, greatly reducing time spent searching for a vertex. The prefix-sum and CSR formats, in contrast, need either much more memory to do in-memory search or they must search on external storage, which is far slower.

4.1.2 Example

We illustrate degree-ordered storage using the graph in Figure 2a. Table 2 shows this graph’s adjacency list. The maximum ID in the original graph is 12, although there are only 7 vertices – a typical scenario in real-world graph data [11, 28–30, 62].

We first sort the vertices by descending out-degree, with ties broken randomly. Table 3
Table 3: Relabeling of Example Graph

<table>
<thead>
<tr>
<th>src dest</th>
<th>degree</th>
<th>new src id</th>
<th>new dst id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2, 4, 12 3 0</td>
<td>3, 5, 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 1, 2, 10 3 1</td>
<td>2, 4, 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2, 7 2 2</td>
<td>1, 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 2, 10 2 3</td>
<td>4, 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 1 4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 10 1 5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Edge List Stored on External Disk

<table>
<thead>
<tr>
<th>offset</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>edges</td>
<td>3 5 6 2 4 6 1 6 4 6 0 4</td>
</tr>
</tbody>
</table>

shows a sorted order in columns 1–3. The next step relabels the vertices based on this order, as shown in columns 4–5 of the table. Figure 2b shows the relabeled graph. Having relabeled the graph, we store: the map between the old and new IDs (columns 1 & 4 from Table 3), the ordering on the adjacency lists (Table 4), a mapping from degree to the first ID with this degree, and mapping from degree to the first out-neighbor’s offset of the first ID with this degree.

Table 5 is the lookup table mapping degree to the first ID having this degree, called the \textit{ids} table. Instead of storing an index for every vertex, this table stores the smallest ID of the vertices with the same out-degree. Table 6 is the lookup table mapping degree number to the offset of the first id having this degree. We call it the \textit{id offset} table. Combined with the \textit{ids} table, GraphZ stores the edges’ starting offset of the smallest vertex. Then, a simple calculation shows how many bytes to read for this vertex. This storage format trades increased computation (to compute indices) for decreased memory footprint. For out-of-core graph processing this is an easy tradeoff – memory is a much more precious resource than computation.

Also, this storage format is very good for random access in out-of-core graphs—we only need to keep \textit{ids} table and \textit{id offset} table in memory. Since the the graphs are often sparse, these two tables typically take just hundreds of kilobytes. To randomly access a vertex \textit{x}, we do a binary search on \textit{ids} table to find the degree \textit{d} satisfying \textit{ids} table[\textit{d}] \leq x < \textit{ids} table[\textit{d}+1]. This \textit{d} is the out-degree of \textit{x}. The first id that has out-degree of \textit{d} is \textit{ids} table[\textit{d}] . Then look in the \textit{id offset} table to get the offset of vertex \textit{ids} table[\textit{d}]. Finally we compute the offset of vertex \textit{x}, by the formula:

\[ \text{offset} = \text{id offset} table[\textit{d}] + (x - \text{ids} table[\textit{d}]) \times \text{d} \]  

For example, to find the offset of vertex 3, we do a binary search on \textit{ids} table to find the degree of vertex 3 is 2 and the first vertex with out-degree 2 is 2. Then we check the \textit{id offset} table[2] and find vertex 2’s offset is 6. As the degree is 2, vertex 3’s offset is \(6 + (3 - 2) \times 2 = 8\). Since the degree is 2, two edges must be read (at offsets 8 and 9). Finally, the disk is read to get the out-edges 4 and 6.

4.1.3 Implementation

Due to space limitations, we omit some implementation details, but the conversion to DOS only requires one pass per graph using an external k-way merge sort [12, 22, 49] to perform
relabeling and build the indexing data structures, which can then be used for many different computations; i.e., the overhead is easily amortized over multiple graph computations. In fact, this storage format is so compact for real graphs that we advocate it becoming a standard for distributing graphs. Our experimental results indicate that GraphZ’s preprocessing time is less than GraphChi and X-Stream (see Sec. 4.3.2).

### 4.1.4 Analysis of Unique Degrees

We argue that the maximum number of a graph’s unique degrees must be small compared to the number of edges and show empirical evidence of this claim. The results justify the utility of degree ordered storage.

**Claim 1.** Given graph $G = (V, E)$, let $UD$ be the set of unique degrees in $G$. Then: $|UD| \leq 2\sqrt{|E|}$

**Proof.** We divide $UD$ into subsets $UD_1$ and $UD_2$.

- Let $UD_1 = \{d \in UD | d < \sqrt{|E|}\}$. Then, $|UD_1| \leq \sqrt{|E|}$.
- Let $UD_2 = \{d \in UD | d \geq \sqrt{|E|}\}$.
- Let $V_2 = \{v \in V | degree(v) \geq \sqrt{|E|}\}$.
- Function $degree(v)$ returns the degree of vertex $v$.
- Assume $|UD_2| > \sqrt{|E|}$, then

\[
|E| = \sum_{v \in V} degree(v) \geq \sum_{v \in V_2} degree(v) \geq \sum_{v \in V_2} \sqrt{|E|} \\
= |V_2| \times \sqrt{|E|} \geq |UD_2| \times \sqrt{|E|} \\
> \sqrt{|E|} \times \sqrt{|E|} = |E|
\]

Thus we get the contradiction: $|E| > |E|$, so the assumption is wrong. And we have $|UD_2| \leq \sqrt{|E|}$. So

\[
|UD| = |UD_1| + |UD_2| \leq \sqrt{|E|} + \sqrt{|E|} = 2\sqrt{|E|}
\]

So, **Claim 1** is proved. This means that, even in the worst case, the number of unique degrees in a graph is small compared to the number of edges, and thus small compared to the number of vertices.

This analysis backs up the notion that the number of unique degrees in a graph must be small. Additionally, Table 7 shows the number of unique degrees for graphs in the SNAP repository. The results confirm that the number of unique degrees in real world graphs is orders of magnitude smaller than the total number of vertices, demonstrating the potential for degree-ordered storage.
### Table 7: SNAP graph properties.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>1.7M</td>
<td>3.8M</td>
<td>3M</td>
<td>457K</td>
<td>2.4M</td>
</tr>
<tr>
<td>Edges</td>
<td>22M</td>
<td>17M</td>
<td>234M</td>
<td>15M</td>
<td>5M</td>
</tr>
<tr>
<td>Unique degrees</td>
<td>2.0K</td>
<td>0.7K</td>
<td>5.4K</td>
<td>1.7K</td>
<td>1.7K</td>
</tr>
</tbody>
</table>

#### 4.1.5 Analysis of Edge Density Distribution

In addition to reducing the storage requirement for indices, DOS reduces disk I/O accesses by allowing a large number of in-memory updates for messages. When processing graphs out-of-core, vertices are divided into *partitions*—disjoint sets of vertices which can all fit in memory at once. When an edge’s source and destination are within the same partition, the framework can do an in-memory update for the message passed from source to destination. Otherwise, it must flush this message to out-of-core storage to be applied when the destination’s partition is loaded. DOS naturally reduces many cross-partition edges, because of its degree-sorted adjacency lists. To illustrate this phenomenon we count the number of in-partition messages as a function of the top n% of vertices; i.e., those in the first partition for different partition sizes. Thanks to the power-law properties of natural graphs, a large number of edges’ sources and destinations are within the first partition, where vertices has highest degrees, compared to other partitions.

![Figure 3: CDF of in-partition messages as a function of partition size.](image)

Fig. 3 shows the ratio of edges within the top n% of vertices for natural graphs used in our evaluation (see Sec. 4.3.1). The x—axis shows the percentage of vertices in the first partition compared to all vertices, and the y—axis shows the ratio of edges within the first partition compared to all edges. For the three natural graphs: small, medium and large, even when there are 20 partitions—i.e., the top 5% of vertices are in the first partition and the graph is...
20× larger than memory—DOS reduces the messages being flushed to disk by about 20%. And because GraphZ only stores vertices in memory, normally, it does not need that large number of partitions. For the graph medium, when there are about 15% in the first partition, GraphZ already saves about 58% out-of-core messages.

### 4.2 Ordered Dynamic Messages

There is a MsgManager inside GraphZ. The MsgManager has two jobs. First, during updates to the current partition, the MsgManager waits for messages to vertices that are currently on disk and stores them. Second, before an iteration on a partition starts, the MsgManager loads all vertices of the current partition into memory, and calls `apply_message()` for any vertices in the partition which have pending messages.

The MsgManager has a separate buffer for every partition of vertices. While the current partition is being updated, the MsgManager waits for messages and puts the message into the right buffer for its destination.

When GraphZ starts a new partition, the MsgManager flushes the last partition’s vertices back to disk and loads the next partition’s vertices to memory. Then the MsgManager reads messages that were sent to the new partition and applies these messages to vertices. To accelerate this process, it is parallelized. To maintain the ordering guarantees and avoid possible conflicts, we use a mutex pool. Our experiments show using mutexes has minimal influence on elapsed time as contention is low during this period. After all old messages for the new active partition are updated, then Sio retrieves edges from storage and the process begins on the loaded partition.

### 4.3 Empirical Evaluation

This section evaluates GraphZ’s innovations and compares to GraphChi and X-Stream. While GridGraph is newer we do not compare against it for two reasons: (1) handling extremely large graphs is GraphZ’s motivation, but GridGraph produces a runtime failure when it tries to ingest our largest graphs; and (2) GridGraph’s open source release only contains three of the six benchmarks we use to test (BFS, PageRank, and Connected Components). We detail the evaluation platform and benchmarks used and then compare the performance and IO burden of GraphZ, GraphChi, and X-Stream.

#### 4.3.1 Experimental Setup

**Hardware Platform** We test on an Intel i7 2700K (4 cores, 8 hardware threads) with 16 GB of RAM. The system runs CentOS 7 and Linux kernel 3.10.0. There are 3 disks: an internal 250GB HDD, an internal 500GB Samsung 840 Pro SSD and an external 4TB HDD connected by an eSATA cable. We use the 250GB disk for the OS and do experiments on the other two. We configure the machine without any swap partitions – eliminating interference from system memory replacement.
Table 8: Graph Properties.
(M = Million, B = Billion, GB = Gigabytes.)

<table>
<thead>
<tr>
<th>Graph Size</th>
<th>LiveJournal</th>
<th>Friendster</th>
<th>YahooWeb</th>
<th>Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>4M</td>
<td>124.8M</td>
<td>1.4B</td>
<td>3.9B</td>
</tr>
<tr>
<td>Edges</td>
<td>69M</td>
<td>3.6B</td>
<td>6.6B</td>
<td>26.2B</td>
</tr>
<tr>
<td>Size</td>
<td>560MB</td>
<td>27.8GB</td>
<td>60.0GB</td>
<td>224.4GB</td>
</tr>
<tr>
<td>Unique degrees</td>
<td>1.3K</td>
<td>3.1K</td>
<td>2.0K</td>
<td>47.4K</td>
</tr>
</tbody>
</table>

Table 9: Vertex index size executing PageRank.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>small</th>
<th>medium</th>
<th>large</th>
<th>xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi</td>
<td>30.8MB</td>
<td>952.4MB</td>
<td>10.5GB</td>
<td>28.8GB</td>
</tr>
<tr>
<td>GraphZ</td>
<td>43KB</td>
<td>49KB</td>
<td>32KB</td>
<td>758KB</td>
</tr>
</tbody>
</table>

Graph Algorithms  We use 6 benchmarks: Connected Components (CC), Breadth-first search (BFS), PageRank (PR) [26, 39], Single-Source Shortest Paths (SSSP), Belief Propagation (BP) [3, 57], and Randomwalk (RW) [33]. GraphChi comes with CC, PageRank, and Randomwalk, while X-Stream lacks Randomwalk. We implement the missing algorithms. Some combinations of benchmarks and storage cause errors for GraphChi or X-Stream. If we are unable to obtain a particular result, that entry in the charts is blank and it is not included when we compute aggregate statistics.

Inputs  Table 14 shows the basic properties, including number of edges, vertices, and storage requirement, of the 3 natural graphs and 1 synthetic graph (generated according to [60]) used in this study: LiveJournal [61] (small), Friendster [61] (medium), YahooWeb [59] (large), and Sim (xlarge). The small graph easily fits into memory on our test machine. The medium graph is larger than our maximum 16GB RAM capacity. The large graph is almost four times larger than memory. The xlarge graph is almost 14 times larger than memory. We note that these storage sizes are just the memory required to hold the graph structure and simplest data for vertices and edges (4B for each). Individual algorithms may require substantially more memory for storing per-vertex/edge local variables.

4.3.2 Preprocessing And Vertex Index Size

To demonstrate degree ordered storage’s benefits, Table 9 shows the vertex index size for each graph in both GraphChi and GraphZ. X-Stream does not require a vertex index because it always streams edges sequentially off of disk. Clearly, GraphZ’s indices are orders of magnitude smaller compared to GraphChi’s. GraphZ’s compact indices create more room to store actual data (rather than the book-keeping indices) and contribute to better overall performance.

Table 10 shows the preprocessing time for the above graphs on both the HDD and SSD. GraphZ has the lowest preprocessing time, despite its seemingly complicated preprocessing to convert to degree-ordered storage. X-Stream has the algorithmically simplest preprocessing, but it is implemented in Python. If it were implemented in C/C++, it would likely be competitive with GraphZ.
Table 10: Preprocessing time (s).

<table>
<thead>
<tr>
<th>Graphs</th>
<th>small HDD</th>
<th>SSD</th>
<th>medium HDD</th>
<th>SSD</th>
<th>large HDD</th>
<th>SSD</th>
<th>xlarge HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi</td>
<td>18</td>
<td>17</td>
<td>3193</td>
<td>1422</td>
<td>6413</td>
<td>2613</td>
<td>42240</td>
</tr>
<tr>
<td>GraphZ</td>
<td>17</td>
<td>17</td>
<td>3040</td>
<td>1102</td>
<td>5969</td>
<td>2299</td>
<td>29561</td>
</tr>
<tr>
<td>X-Stream</td>
<td>265</td>
<td>258</td>
<td>13701</td>
<td>13402</td>
<td>25046</td>
<td>24809</td>
<td>99124</td>
</tr>
</tbody>
</table>

Figure 4: Run time comparison on the xlarge graph.

4.3.3 Extra Large Graph Performance

The results for the extra large graph (Sim) on HDD are shown in Fig. 4 (the SSD cannot hold this graph). The x-axis shows benchmarks, while the y-axis shows the execution time (in kiloseconds – lower represents improved performance). These results demonstrate that GraphZ achieves significantly lower run times than X-Stream. Unfortunately, GraphChi does not work for such a large graph on our test system because GraphChi’s vertex index does not fit into memory (see Table 9). The harmonic mean of speedup shows GraphZ is 1.86× faster than X-Stream. GraphZ’s maximum speedup is 3.06× compared to X-Stream on RW. We do not include a data point for X-Stream on BP in these results because the per-vertex data for this algorithm on this graph makes it too large for our experimental system to handle.

4.3.4 Large Graph Performance

The results for the large graph (YahooWeb) are shown in Fig. 5(a). This figure has 12 charts, the left column showing results with the magnetic disk and the right column showing SSD results. Each row corresponds to a different benchmark. The x-axis shows the amount memory used in the benchmark, while the y-axis shows the execution time (in kiloseconds – lower represents improved performance).

These results demonstrate that GraphZ achieves significantly lower run times than either GraphChi or X-Stream. For the HDD, the harmonic mean of speedup shows GraphZ is 4.84× faster than GraphChi and 3× faster than X-Stream. GraphZ’s maximum speedup is 8× compared to GraphChi on SSSP and 7.5× compared to X-Stream on RW. For the SSD, the harmonic mean of speedup shows GraphZ is 1.80× faster than GraphChi and 1.85× faster than X-Stream. The maximum speedup of GraphZ is 3.7× compared to GraphChi on BFS and 13× compared to X-Stream on RW. All approaches benefit tremendously from moving to SSD. GraphZ still provides a significant performance gain, however.
Figure 5: Run times for different graph sizes.

4.3.5 Medium Graph Performance

Fig. 5(b) shows results for the medium graph. The layout of this figure is the same as that for the large data. The relative performance difference between GraphZ and the other packages is even larger in this case. Using harmonic mean for the HDD, GraphZ runs about $7.3 \times$ faster than GraphChi and $8.3 \times$ faster than X-Stream. GraphZ’s maximum speedups are $33 \times$ compared to GraphChi on BFS and $50 \times$ compared to X-Stream on RW.

Using harmonic mean for the SSD, GraphZ is $2.3 \times$ faster than GraphChi and $3.2 \times$ faster than X-Stream. The maximum speedup is $9.5 \times$ for GraphZ compared to GraphChi on SSSP and $71 \times$ for GraphZ compared to X-Stream on RW.

4.3.6 Small Graph Performance

The results for the small graph are shown in Fig. 5(c). The layout of this figure is the same as the previous two. At this small size, we do not see a clear best solution. The fastest graph package varies from benchmark to benchmark. Also, not surprisingly, the type of the backing
store does not meaningfully change the results.

For small graphs like LiveJournal, optimizations for in-memory processing are very important. Since GraphZ’s focus is on improving the performance of out-of-core processing, our current implementation does not have many in-memory optimizations. In addition, because of GraphZ’s deep pipeline overhead, we can expect some slow down on small graphs. Fig. 5(b), however, shows the results of in-memory graph processing with GraphZ are competitive or even sometimes much better than existing approaches.

4.3.7 Performance Breakdown

![Figure 6: Performance breakdown for the large graph.](image)

We analyze each of the proposed technique’s contributions to performance in Fig. 6. The x-axis shows benchmarks, while the y-axis shows the execution time (in kiloseconds – lower represents improved performance) for the large graph. All results use the SSD. For each benchmark, the chart shows the runtime for GraphChi, for GraphZ with DM disabled and without using DOS, for GraphZ with DM and without DOS, and for the full GraphZ implementation with both DOS and DM.

These results show that the GraphZ engine without DOS or dynamic messages is actually slower than GraphChi in many cases. In fact, most of GraphZ’s performance improvement comes from DOS: the harmonic mean of speedup shows that full GraphZ is $1.94 \times$ faster than GraphZ without DOS. GraphZ’s maximum speedup is $2.54 \times$ compared to GraphZ without DOS on BP.

For dynamic messages, the harmonic mean of speedup shows GraphZ without DOS is $1.10 \times$ faster than GraphZ without DOS and DM. The maximum speedup of GraphZ without DOS is $1.89 \times$ compared to X-Stream on CC. For algorithms like BFS and SSSP, which produce fewer messages than PR, GraphZ without DOS and DM is even faster than GraphZ without DOS. The reason is that DM incurs more computation overhead and sometimes blocks disk IOs.

4.3.8 Bulk vs. Asynchronous Execution

Graph processing is inherently iterative. Each framework (GraphChi, X-Stream, and GraphZ) continually iterates over the vertex space. GraphChi adopts an asynchronous execution model to reduce the total number of iterations compared to the bulk synchronous model – which is used in X-Stream. GraphZ also adopts the asynchronous model. In this section we compare the models by measuring the number of iterations required for convergence.

Table 11 shows the iterations each approach requires for convergence for three algorithms on both the LiveJournal and Friendster graphs, for the omitted graphs GraphChi and GraphZ.
Table 11: Iterations for Convergence

<table>
<thead>
<tr>
<th></th>
<th>SSSP</th>
<th>CC</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphChi</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>X-Stream</td>
<td>47</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>GraphZ</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphChi</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>X-Stream</td>
<td>59</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>GraphZ</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 7: Total IO volume for large graph

achieve the same iteration counts. Since GraphChi and GraphZ both use the asynchronous model, they require significantly fewer iterations than X-Stream. This difference is a key factor in performance difference between GraphZ and X-Stream.

It is also important to note that these results show that degree ordered storage does not reorder the vertices in a way that achieves advantageous iteration counts. In fact, GraphZ’s iteration counts are sometimes higher than GraphChi’s. These results demonstrate both the advantages of the asynchronous model, and also provide evidence that degree ordered storage provides a real benefit and the speedup is not due to some advantageous vertex ordering compared to GraphChi.

4.3.9 IO Statistics

Throughout the paper, we argue that GraphZ’s degree-ordered storage and dynamic messages greatly reduce the IO burden, and the reduced IO leads to reduced runtime. We have demonstrated reduced run time, in this section we evaluate the IO operations explicitly. Fig. 7 compares the exact external IO of the three graph engines on two algorithms: PageRank and BFS with the large graph (YahooWeb). These results are representative of the IO statistics for all algorithms; the others are omitted for space. The figure shows that for PageRank, GraphZ performs less than half the reads of GraphChi and less than one third that of X-Stream. When running BFS, GraphZ needs just less than a third of the reads of GraphChi and X-Stream. This data confirms that GraphZ’s model provides tremendous IO reduction compared to other state-of-the-art approaches.
5 GraphStone: Hybrid Model and On the fly vertex assembly

5.1 Hybrid Execution Model

We describe GraphStone’s execution model, which combines properties of the BSP and ASYN models. Fig. 8 shows the vertices exposed to vertex $v$ at iteration $i$, for BSP, ASYN and GraphStone’s HYBRID model, given a graph $G = (V, E)$. $v$ is marked with blue to show the runtime is currently calling the update() method on vertex $v$, while vertices in orange are the values that can be seen during this update.

Let $v^i$ be the value of vertex $v$ at iteration $i$. We define $VE(v, i)$ as the vertex values exposed to $v$ at iteration $i$. As shown in Fig. 8a, the set $VE_{BSP}(v, i)$ for BSP is:

$$VE_{BSP}(v, i) = \{ x^{i-1} | x \in V \}$$

For ASYN, we divide vertices set $V$ into 2 parts: $V_1$ and $V_2$. For $\forall i \in V_1$, we have $i < v$ and for $\forall i \in V_2$, that $i \geq v$. As shown in Fig. 8b, $v$ can access the values of vertices in iteration $i$ in $V_1$ and vertices values in iteration $i – 1$ in $V_2$ in iteration $i$. The set $VE_{ASYN}(v, i)$ is:

$$VE_{ASYN}(v, i) = \{ if(x \geq v) x^{i-1} else x^i | x \in V \}$$

Section 3.4 argues that BSP is easier to parallelize than the ASYN model. The arrows shown in Fig. 8 provide examples of updates that could occur in parallel in each model. GraphStone’s HYBRID model provides access to slightly fewer (normally < 5%) of the most recent vertex updates compared to ASYN in exchange for increased parallelism compared to BSP. Specifically, GraphStone partitions vertices into $np$ partitions, each with $W$ vertices. We define $p(v)$
to get the partition id of vertex \( v \). Set \( \mathcal{E}_{HYBRID}(v, i) \) is then:

\[
\mathcal{E}_{HYBRID}(v, i) = \{ \text{if } (p(x) \geq p(v)) x^{i-1} \text{ else } x^i \mid x \in V \}
\]

When \( W = |V| \), the HYBRID model is equivalent to BSP, and when \( W = 1 \), the HYBRID model is the ASYN model. So any algorithm that works under either BSP and ASYN will also work in the HYBRID model. Appendix A provides additional formalism arguing that for a reasonable number of partitions, the HYBRID model achieves iteration counts very close to ASYN for random graphs.

### 5.2 Partitions in GraphStone

The key to GraphStone is that it is extremely parsimonious with memory, allocating just enough to hold the current partition. GraphStone spends extra computational time to reduce memory footprint, often reassembling the same data structures in separate iterations, but given that memory is the most precious resource for large-scale Graph analysis, we find this to be a worthwhile tradeoff.

Fig. 9 and Algorithm 1 illustrates the key pieces of the GraphStone implementation. The GraphStone runtime creates a *patch* which is simply scratch space in memory that is big enough to hold the current partition; ideally, the partitions fit into on-chip last-level cache. The runtime allocates key graph data structures with `mmap`, so the kernel automatically loads and evicts pages without requiring explicit calls from the GraphStone runtime. The GraphStone runtime also allocates worker threads equal to the number of cores on the machine.

When processing the first partition, the workers first copy vertices’ value and scheduling status to the *patch*. It then expands vertices from their compact *in-storage* form to their larger *full* form. The in-storage form only contains the user-defined vertex data type, degrees and the activation status. The full form contains the in-storage data and both the in- and out-edge lists. After expanding the vertices, the runtime schedules workers to invoke the `update()` method on active vertices. After all updates are finished, the runtime copies the updated vertices back to their in-memory form. This process repeats for all subsequent partitions in the iteration.
Iterations continue until a user-specified limit is reached, there are no more active vertices, or the user has signaled that no further iterations are required.

5.3 On-the-fly Vertex Assembly

First, we define edge metadata, which is used to assemble full vertices on-the-fly from in-storage vertices. Because we have two forms of vertices: in-storage and full vertices, we need edge metadata as descriptions to assemble edge lists and convert compact in-storage vertices into their full representation. And in general, these edge metadata is several times larger than vertices, which is the main motivation why we separate edge metadata from full vertices.

Fig. 10 shows the difference when GraphStone processes graphs in memory or out-of-core. There are three distinct components that must be managed: the in-memory vertices, the edge metadata, and the edges themselves. GraphStone uses standard memory allocation to make storage space large enough to hold all vertices in their in-storage format. Even for the largest graphs in our test set, the vertices always fit into memory. For the edge metadata and edges, GraphStone always maps them. Memory mapping these large data structures makes the kernel manage the location of these structures. When the GraphStone runtime tries to access data that is not in memory, the kernel automatically does the page swap. Similarly, when system RAM is close to full, the kernel automatically evicts some pages. In this way, the GraphStone runtime does not even need to implement its own buffering system.

In summary, if the system RAM is large enough to hold the vertices, edge metadata, and edges, then GraphStone does in-memory processing automatically. Otherwise, Graphstone is computing out-of-core. Note that the runtime does not need to detect this, it happens automatically as a result of the implementation. This implementation is aided by having the user define the gen_msg() method—which generates messages on the fly rather than requiring additional storage space for messages—and by the on-the-fly vertex construction—which trades the time to reconstruct full vertices for each partition for decreased memory usage. The only difference in the GraphStone runtime between in-memory and out-of-core processing is that if the edge metadata and edge list are larger than system memory, and it allocates additional threads to overlap computation and I/O for out-of-core processing.

From this implementation, all processing can be done in parallel, including copying vertices’ values to the patch, constructing the full vertices, calling update() on every vertex in the patch, and copying updated vertex values back to the global array. The only synchronization occurs after copying in-memory vertices to patch, each patch is updated and again when each patch is copied back to memory. Under in-memory mode, those synchronizations are all done in memory, so the overhead is very small. Under out-of-core mode, the overhead is also small in comparison with the cost of copying the updated vertex values back to disk.

5.4 Shuffling Graphs Without a Lookup Table

Sec. 3.4 shows how iteration count can be affected by the longest path in a graph. Specifically, if the longest path is in the opposite direction of vertex traversal, the asynchronous model will need the same number of iterations as the bulk synchronous model. We therefore introduce graph shuffling to reorder vertex numbers, reduce the length of the longest path in a partition,
and thus decrease the number of iterations required for convergence. Because memory is a precious resource in graph computation, we want to shuffle the graph without requiring an additional lookup table. Therefore, we introduce a graph shuffling method which requires no additional lookup table.

The main idea is to do a matrix transpose to the vertex arrays. Let us take Fig. 1(a) as an example. First we choose a parameter $n_{row}$, to indicate the number of rows in the matrix we want to use. For Fig. 1(a), we choose $n_{row}$ to be 2. Table 12a shows that we first arrange the vertices in a Matrix of $2 \times 4$, because we have only 7 vertices in total, so we add an extra zero-degree vertex here. In table Table 12(b), we do a simple transpose to the matrix. Next, we relabel all vertices in the row-order. Table 12(c) shows the mapping relationship between the new vertex id and original id. In reality, we simply choose $n_{row}$ from 7-17, so one need not worry that this method may add too many extra vertices, because the upper bound on extra vertices is $n_{row} - 1$.

Algorithm 2 shows the mapping functions between the new id in shuffle order and the original id. The mapping function is computationally simple and has almost zero memory footprint. This function is called once before any iterations begin and so the overhead is very small.

Table 13 shows the iteration number requirement before and after shuffling, with 4 parti-
Algorithm 1 Engine Workflow

1: Global Variables:
2: $np; \triangleright$ number of partitions
3: $max_{iter}; \triangleright$ max number of iterations
4: need_more_iteration; \triangleright$ mark whether more iterations are required
5: function $\text{ITERATE}\_\text{ONCE}(\text{cur_iter})$
6: while $\text{cur_par} < \text{max_par}$ do
7: for $\text{partition} \in \text{partitions}$ do
8: parallel for In-memory Vertex $v \in \text{partition}$ do
9: if $v$ is activated then
10: construct corresponding Full Vertex $v'$ of $v$ in patch
11: copy value and scheduling status of $v$ to $v'$
12: mark $v$ as inactivated
13: end parallel for
14: parallel for Full Vertex $v' \in \text{patch}$ do
15: if $v'$ is activated then
16: update vertex $v'$
17: all activation for other vertices made during update directly goes to in-memory vertices, so there is no race
18: end parallel for
19: parallel for Full Vertex $v' \in \text{patch}$ do
20: if $v'$ is activated then
21: copy value of $v'$ back to corresponding In-memory Vertex $v$ in partition
22: mark $v'$ as inactivated
23: end parallel for
24: function $\text{START}\_\text{ENGINE}$
25: $\text{cur_iter} \leftarrow 0$
26: while $\text{cur_iter} < \text{max_iter}$ and need_moreIteration do
27: need_moreIteration $\leftarrow$ false
28: $\text{ITERATE}\_\text{ONCE}(\text{cur_iter})$
29: $\text{cur_iter} \leftarrow \text{cur_iter} + 1$

5.5 Empirical Evaluation

We demonstrate that GraphStone achieves best-in-class performance for both in-memory and out-of-core processing.

5.5.1 Experimental Setup

Hardware Platform We test on a Linux x86 server with two Intel Xeon E5-2690 processors, 64GB RAM, a 500GB HDD, a 2TB HDD, a 5TB HDD and a 1.2TB SSD. We use the 500GB HDD for the OS. All experiments are conducted using the SSD to store the graphs.
Table 13: Iteration Number after Random Shuffling

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Original Founded</th>
<th>Shuffled Founded</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0,1,2</td>
<td>0,2,4,6</td>
</tr>
<tr>
<td>2</td>
<td>0,1,2,3,4</td>
<td>0,2,4,6,1,3,5</td>
</tr>
<tr>
<td>3</td>
<td>0,1,2,3,4,5,6</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Algorithm 2 Mapping Function from Shuffled ID to Actual ID

1: **Global Variables:**
2:    $size \rightarrow nrow$;
3:    $size \rightarrow ncol$;
4: **function** MAP2NEWID(oldid)
5:    return $oldid \% nrow \times ncol + oldid/nrow$
6: **function** MAP2NEWID(newid)
7:    return $newid \% ncol \times nrow + newid/ncol$

Graph Algorithms  We use 5 algorithms common to most graph evaluations: PageRank(PR) [26], Randomwalk(RW) [33], Breadth-First Search(BFS), Connected Components(CC), and Single-Source Shortest Paths(SSSP). Ligra comes with a rather slow CC, so we implement a much faster one for it (more than 30×). Ligra does not come with an SSSP or RW implementation, so we implement those ourselves in Ligra. GraphChi’s distribution does not contain BFS and SSSP implementations, so we implement those ourselves. Similarly, we implement RandomWalk for X-Stream. Ligra users need to make a decision between either using atomic operations or allowing some small error/delay. We always choose the fastest alternative after experimenting. GraphStone is set so that the partitions of in-memory graphs have 200K vertices and out-of-core graphs have 1M. Also, for each algorithm, framework, and graph we run 5 times and report the mean runtime.

Graphs  For in-memory testing, we use graphs from the original Ligra paper and SNAP datasets[48]: rm24, rm27, twitter [24] and lj [61]. rm24 and rm27 are generated graphs [5, 45], while the other two are natural graphs. For the 2 small graphs from the original Ligra paper that we do not use, the graph generator is not provided, so we do not have access to them. For out-of-core engines, we use 3 graphs: sim13, sim26 and the Common Crawl Graph (CCG), which is the largest publicly available natural graph [10, 41]. sim13 and sim26 are generated by an improved power-law generator that mimics natural graphs [60]. Table 14 shows the properties of those graphs.

5.5.2 Best-in-Class Performance

We briefly highlight summary results that back our main claim: GraphStone out-performs best-in-class approaches for both in-memory and out-of-core processing.

Fig. 11 summarizes all our results, with in-memory results compared in Fig. 11(a) and out-of-core results compared in Fig. 11(b). Fig. 11(a) shows speedups over Ligra (the best
Table 14: Graph Properties.

(M = Million, B = Billion.)

<table>
<thead>
<tr>
<th>Graph</th>
<th>rm24</th>
<th>rm27</th>
<th>twitter</th>
<th>lj</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>1.7M</td>
<td>33M</td>
<td>42M</td>
<td>4.0M</td>
</tr>
<tr>
<td>Edges</td>
<td>99M</td>
<td>212M</td>
<td>1.5B</td>
<td>6.6B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph</th>
<th>sim13</th>
<th>sim26</th>
<th>CCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>2B</td>
<td>3.9B</td>
<td>1.7B</td>
</tr>
<tr>
<td>Edges</td>
<td>13B</td>
<td>26B</td>
<td>64B</td>
</tr>
</tbody>
</table>

Figure 11: Summary data showing GraphStone’s speedups over best in class frameworks for In-Memory (a) and Out-of-Core (b) processing.

prior in-memory framework) for a serial, hand-tuned C program, X-Stream, and GraphStone. Fig. 11(b) shows speedup over X-Stream (the best prior out-of-core framework). Neither Ligra nor the hand-tuned C code work out-of-core, so Fig. 11(b) does not show results for these two approaches.

Fig. 11(a) shows that Ligra is $3 \times$ faster than a hand-tuned C program and almost $15 \times$ faster than X-Stream. Ligra’s parallelism allows it to out-perform-the serial C-code. X-Stream has very high overhead due to the book-keeping data structures that allow it to work out-of-core. GraphStone, however, is $2.5 \times$ faster than Ligra, making it tremendously faster than the C code and X-Stream for in-memory processing.

Fig. 11(b) shows that GraphStone is $6.5 \times$ faster than X-Stream when working out-of-core. These results back up the central claim made in the introduction: prior to GraphStone, graph analysts had to choose between fast in-memory frameworks, or an out-of-core framework that is slow on small graphs, but works on large graphs. Now—with GraphStone—analysts can be assured of best-in-class performance regardless of the relationship between graph size and memory. The remainder of this section presents specific results and details demonstrating how GraphStone achieves these results.

5.5.3 In-memory Experimental Results

This section shows run time comparisons for in-memory processing and detailed analysis for the main reasons that GraphStone out-performs prior approaches.

Run Time Comparisons Fig. 12(a) shows the run time comparisons between Ligra and GraphStone for twitter. Each row corresponds to a benchmark. The bar heights represent run time. For this graph, GraphStone is faster than Ligra for every algorithm, especially for those...
that perform dense traversal—\textit{i.e.}, PR and RW where most vertices are accessed for most iterations. GraphStone’s harmonic mean speedup over Ligra is about 3.9×. The maximum speedup is 6.8×, for PageRank. Ligra is especially optimized for sparse traversal algorithms—\textit{i.e.}, BFS, CC, and SSSP, where only a few vertices are accessed on most iterations—and its performance on those algorithms is especially good. Despite Ligra’s focus on such algorithms, GraphStone still outperforms it.

Fig. 12(b) shows the runtime comparisons on graph \textit{rm27}. GraphStone achieves a harmonic mean speedup of 3.8× over Ligra. The maximum speedup is 5.0×, achieved on PR. Fig. 12(c) shows the runtime comparisons on graph \textit{rm24}. For this graph, the sparse traversal algorithms finish in about 2 seconds. GraphStone’s harmonic mean speedup is about 1.4× over Ligra. The highest speedup is 2.0× on PR. Because \textit{rm24} is a rather small graph, we find the runtime differences for the same algorithm may be as high as 0.5 seconds. That’s why we run them for 5 times and get the average. Fig. 12(d) compares runtimes for the LiveJournal graph. GraphStone’s harmonic mean speedup is 2.5× that of Ligra, with a maximum speedup of 2.9× on PageRank.

For completeness, we include in-memory performance for both X-Stream (in Table 15) and a hand-tuned serial C code Table 16. Both approaches are much slower than Ligra and GraphStone, so we do not include them in the charts as doing so would skew the results. Just comparing the run times in the tables to the scale of the charts in Fig. 12 shows that both Ligra and GraphStone far outperform these approaches on in-memory data.

\textbf{Memory Bandwidth Comparisons} Fig. 13(a)-(d) shows memory bandwidth utilization for Ligra and GraphStone for \textit{twitter} (a), \textit{rm27} (b), \textit{rm24} (c), and \textit{LiveJournal} (d). These results show that GraphStone achieves a much better memory bandwidth utilization than Ligra on aver-
Table 15: Run times (s) for X-Stream.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>twitter</th>
<th>rm27</th>
<th>rm24</th>
<th>lj</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>447</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>CC</td>
<td>534</td>
<td>43</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>BFS</td>
<td>202</td>
<td>42</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>SSSP</td>
<td>656</td>
<td>45</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>RW</td>
<td>1667</td>
<td>248</td>
<td>144</td>
<td>123</td>
</tr>
</tbody>
</table>

Table 16: Run times (s) for hand-tuned C code.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>twitter</th>
<th>rm27</th>
<th>rm24</th>
<th>lj</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>687</td>
<td>151</td>
<td>84</td>
<td>26</td>
</tr>
<tr>
<td>CC</td>
<td>56</td>
<td>12</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>BFS</td>
<td>51</td>
<td>12</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>SSSP</td>
<td>56</td>
<td>13</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

The higher achieved memory bandwidth is key to GraphStone’s performance advantage as getting data into and out of the processor is the main bottleneck for these graphs and algorithms, especially the larger graphs.

### 5.5.4 Out-of-core Experimental Results

This section explores performance on graphs that must be processed out-of-core.

**Run Time Comparison**  
Fig. 14 shows the runtime for the 3 large graphs running out-of-core. GraphStone completes all algorithms for all three graphs. In some cases, GraphStone is so much faster than X-Stream that it looks like there is no result. In contrast, X-Stream does not complete for every graph and algorithm. For charts where there is no X-Stream bar, X-Stream exited with either an out-of-memory or out-of-disk-space error. If X-Stream failed to complete for a graph and algorithm we do not include that point in the calculation of mean speedup.

Fig. 14(a) shows the runtime for the `ccg` graph, which is the largest graph we use. Unfortunately, X-Stream failed on PR and CC. Excluding those points, GraphStone achieves a speedup of $13.8 \times$ over X-Stream. The biggest speedup is achieved on RW which is $60.6 \times$.

Fig. 14(b) shows the runtime for the `sim26` graph, which is the second largest graph we use. By harmonic mean, GraphStone gets a speedup of $13.1 \times$ over X-Stream. The biggest speedup is achieved on RW with $45.1 \times$ over X-Stream.

Fig. 14(c) shows the runtime for `sim13` graph, which is the third largest graph we use. By harmonic mean, GraphStone gets a speedup of $3.2 \times$ over X-Stream. The biggest speedup is achieved on RW with $26 \times$ over X-Stream.

Please note that, for GraphStone, CC and SSSP spend more time than PR for a single iteration because we may load the same block multiple times by random access within a single iteration. The users could change a simple setting to enable all sequential access, then CC and SSSP would have similar performance as PR. To give readers scientific understanding of how the results look, we do not enable this setting. Otherwise, the results for GraphStone would be
even better.

From the above, we see that when the graph is large enough, GraphStone achieves much better performance than X-Stream. When the graph is not that big X-Stream is more competitive, but GraphStone is still several times faster than X-Stream.

**I/O Usage Comparisons**  Fig. 15 compares the I/O usage for the 3 large graphs running out-of-core. Fig. 15(a) shows the I/O usage for the *ccg* graph. By harmonic mean, X-Stream makes about \(5.8\times\) the I/O access over GraphStone. The biggest I/O reduction is achieved on BFS with \(32.3\times\). Fig. 15(b) shows the I/O usage for the *sim26* graph. By harmonic mean, X-Stream uses about \(4.1\times\) as much I/O as GraphStone. The biggest I/O difference with X-Stream is achieved on SSSP with \(5.4\times\). Fig. 15(c) shows the I/O usage for the *sim13* graph. By harmonic mean, X-Stream makes about \(3.5\times\) as many I/O access as GraphStone. The biggest I/O difference with X-Stream is achieved on BFS with \(5.9\times\). One of the reasons that GraphStone outperforms X-Stream is that it makes many fewer IO requests. Because GraphStone uses on-the-fly vertex generation, it can store all vertices in memory. Thus GraphStone does not need to store messages to disk, which dramatically reduces its required IOps compared to X-Stream.

**I/O Bandwidth Comparison**  Fig. 16 compares achieved I/O bandwidth for the 3 large graphs running out-of-core. Fig. 15(a) shows the I/O bandwidth for *ccg*. Fig. 15(b) and Fig. 15(c) compared the I/O bandwidth for *sim26* and *sim13*. GraphStone has similar I/O bandwidth utilization as X-Stream. For the relatively small *sim13* graph, GraphStone keeps a lot of its data in cache and does not need to read a lot from disk so its bandwidth is lower than X-Stream. With Fig. 16, we conclude that GraphStone’s execution model achieves better I/O bandwidth compared to BSP and edge-centric systems—like X-Stream—while having the high usability as ASYN and vertex-centric systems. And the vertex-centric interfaces could have on-the-fly newest values for vertices, allowing us to develop new execution models to speedup
5.5.5 Convergence Speed Study

We argue that the HYBRID synchronous model proposed in GraphStone provides faster convergence speed than BSP. In this section we compare the convergence speed for the two natural graphs that can be processed in memory. Specifically we compare Ligra’s iteration counts (recall Ligra uses the BSP model) to GraphStone’s with different partition numbers and to hand-tuned serially C code that is fully asynchronous (the same code used in Fig. 11).

The results are shown in Table 17 for twitter and in Table 18 for lj. The HYBRID models are parameterized by the number of partition: HYBRID – n means there are n partitions. The results clearly indicate that with even a small number of partitions, GraphStone requires far fewer iterations than Ligra’s BSP model and is generally competitive with the fully asynchronous code, but achieves much better parallelism.
6 Approximate in-memory compression and computation

6.1 Lossy Graph Compression

Large graphs often cannot fit in memory, so we have to use out-of-core systems for large graphs. But out-of-core systems are often much slower compared to in-memory systems. There are two main methods for out-of-core systems: (1) Using external storage and (2) using distributed systems. For (1), though there are some extreme fast SSDs available, but the I/O performance are still far less compared to DRAM. And most SSDs are accessed by blocks, which cause the random I/O performance to be much lower than DRAM. For (2), those the graph could reside in memory, and the communication between nodes have high overhead. Also, overhead of data sharing and synchronization in the distributed systems are much higher than a single machine.

This thesis presents another way for handling large graphs exceeding the memory, with slightly less accurate and lossy in-memory compression. This involve two major techniques: vertex compression and edge compression:

- **Vertex Compression** Given two edges $< v_1, v_2 >$ and $< v_2, v_3 >$, and $\text{deg}(v)$ gives the out degree of $v$. If we don’t concern about $v_2$, we could compress these 2 edges as a new edge $< v_1, v_3, \text{deg}(v_1) \cdot \text{deg}(v_3), 2 >$, or simply $< v_1, v_3 >$. By this way we could heavily reduce vertices storing in memory while $\text{deg}(v_1) \cdot \text{deg}(v_3)$ and 2 still describe familiarity and distance between $v_1$ and $v_3$. For algorithms like Dijkstra, we could do more compression,
that we only need the distance between these 2 vertices. For algorithm like Pagerank, if $v_2$ is a low degree vertex, we could delete these 2 edges.

- **Edge Compression** For some algorithms, we don’t need all edges. And we could simply delete some edges, without influence major property of the graph. We could do some random edge samplings here. Also, given an edge $<v_1, v_2, familiarity, distance>$. If familiarity is very small, we could ignore it for Pagerank algorithm, then we could simply delete it from the graph. For Dijkstra, we could simply delete this edge and update it to vertices.

By above methods, we could save a lot of memory usage and speedup computation for each iteration because of less workload. And during the compression, though we delete a lot of vertices and edges, we still preserve a lot of information.

### 6.2 Algorithms

We decide to use some common algorithms, Connected Components (CC), Breadth-first search (BFS), PageRank (PR), Single-Source Shortest Paths (SSSP), k-Means (KM), Belief propagation (BP), Triangle Counting (TC) and Randomwalk (RW). In Table 19, we show the metrics for measuring accuracy of approximated version of those algorithms. On choosing those metrics, we normally stand in the end users’ view that how the results would be used, and then we see the influence of errors.

### 6.3 Preview Results

In this section, we would show some preview results using some lossy compression methods. We use graph $l j$ as the input in the DOS format. TC is a common method to compute clustering coefficient in graph. For TC, we also get the accurate number, and then compare it with results.
Table 19: Metrics for measuring accuracy of algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Ratio of misplaced vertices</td>
</tr>
<tr>
<td>BFS</td>
<td>Ratio of vertices failed to be founded</td>
</tr>
<tr>
<td>PR</td>
<td>Ratio of vertices failed to be founded in top k</td>
</tr>
<tr>
<td>SSSP</td>
<td>Ratio between L1-norms of errors and distances</td>
</tr>
<tr>
<td>KM</td>
<td>Ratio of misplaced vertices</td>
</tr>
<tr>
<td>RW</td>
<td>Ratio of vertices failed to be founded in top k</td>
</tr>
<tr>
<td>TC</td>
<td>Ratio of estimated triangle number over exact triangle number</td>
</tr>
<tr>
<td>BP</td>
<td>Ratio of different labels of vertices</td>
</tr>
</tbody>
</table>

from compressed graphs. PR and RW are used for analyzing graphs and find important vertices. On measuring the accuracy, we use $top - k$. We first use the original version of PR and RW to produce accurate results. Then we use part of the graph, and see how many $top - k$ vertices in the accurate result are found in the approximate results.

Table 20: Accuracy of Triangle Counting on graph $lj$

<table>
<thead>
<tr>
<th>Percent of Graph Used</th>
<th>100%</th>
<th>50%</th>
<th>25%</th>
<th>12.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Triangles</td>
<td>17720130</td>
<td>174713488</td>
<td>173275328</td>
<td>167078912</td>
</tr>
<tr>
<td>Error</td>
<td>0%</td>
<td>-1.7%</td>
<td>-2.6%</td>
<td>-6%</td>
</tr>
</tbody>
</table>

Table 20 shows the approximate graph computing results for triangle counting. If we use 1/8 of the graph, we still get about only 6% error. But the speed is more than 20× faster than using the whole graph.

Table 21: Accuracy of Pagerank on graph $lj$

<table>
<thead>
<tr>
<th>Percent of Graph Used</th>
<th>Top-10</th>
<th>Top-100</th>
<th>Top-1000</th>
<th>Top-10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>100%</td>
<td>96%</td>
<td>95.8%</td>
<td>93.6%</td>
</tr>
<tr>
<td>25%</td>
<td>90%</td>
<td>94%</td>
<td>90.5%</td>
<td>88.3%</td>
</tr>
<tr>
<td>12.5%</td>
<td>90%</td>
<td>89%</td>
<td>85.3%</td>
<td>81.7%</td>
</tr>
</tbody>
</table>

Table 21 shows the approximate graph computing results for PR. The column ”Percent of Vertices Used” shows that how many vertices are used in the computation, and columns ”top-k” shows how many top$k$ results are successfully founded. From the table, we could see that, even if we use 1/8 of the graph, we could still get about 80-90% accuracy. For an application like search engine, this error is unnoticeable.

Table 22 shows the approximate graph computing results. The result is even better than Pagerank. From the table, we could see that, even if we use 1/8 of the graph, we could still get more than 90% accuracy for top 10 – 1000.

7 Future Plan

The current approximate results looks very close to the exact results. Here is our plan for future:
Table 22: Accuracy of Randomwalk on graph 1j

<table>
<thead>
<tr>
<th>Percent of Graph Used</th>
<th>Top-10</th>
<th>Top-100</th>
<th>Top-1000</th>
<th>Top-10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>100%</td>
<td>99%</td>
<td>94.7%</td>
<td>77.7%</td>
</tr>
<tr>
<td>25%</td>
<td>100%</td>
<td>99%</td>
<td>94%</td>
<td>72.6%</td>
</tr>
<tr>
<td>12.5%</td>
<td>100%</td>
<td>99%</td>
<td>92.7%</td>
<td>72.6%</td>
</tr>
</tbody>
</table>

- Get major interfaces done and get experimental results for 6-10 algorithms. (by Dec 2017)
- Study how the accuracy change as compression ratio change. And is it possible predict the target accuracy with certain compression ratio. (by Feb 2018)
- Study how the accuracy change as graph change. And see the influence of graph properties on accuracy. (by Apr 2018)

## A Convergence in the HYBRID Model

### A.1 Path Iteration Length

To help the readers understand convergence speed, we define **Path Iteration Length**: *Under a specified iteration method, for a given path, the number of iterations required for data generated from the start vertex to reach the end vertex.* In general, the shorter the **Path Iteration Lengths**, the more efficiently messages are passed, and the faster the graph computation converges.

For bulk synchronous processing (BSP), since no vertex can see other vertices’ values generated in the current iteration, data on every path can only move by 1 step every iteration, thus, the path iteration length is the same as the path length for any given path. The longest path in Fig. 1(a) is 0–6, so its path iteration length is also 6. But for ASYN on Fig. 1(b), if we iterate from vertex 0–6, then path iteration length of path 6–0 is 6, because it takes 6 iterations to pass the mark from vertex 6 to vertex 0, but if we traverse from vertex 6–0, then the path iteration length of path 6–0 is 0, because at the end of iteration 1, vertex 6 already sees vertex 0’s mark. The initial stage is an extra iteration, but we count message passing starting from iteration 1, so the path iteration length of path 6–0 is 0.

Another issue here is the parallelism. For BSP, vertex values in iteration $i$ only rely on vertices’ values from iteration $i − 1$, so a framework could allocate as many threads as the number of vertices when iterating. Within an iteration, All vertex updates are independent. But for ASYN—taking Table 1(a) as an example at iteration 1—to update vertex 5, a framework needs to wait for updates to vertex 4. This dependence chain goes all the way to vertex 0. Such dependences mean vertices are updated serially. This example does not mean ASYN cannot be parallelized at all. In Table 1(b), there are no in-edges from vertex 0 to 1, so vertex 0 and vertex 1 could be updated at the same time. There are some methods to parallelize the ASYN model; e.g., using topological sort to find independent vertices. Unfortunately, such methods have high cost and are very difficult to load balance.
A.2 Convergence Speed Analysis

In this section, we use Path Iteration Length (PIL) to analyze the convergence speed for the iteration methods: BSP, ASYN, and HYBRID execution models on a random path. Without loss of generality, we use a random graph $G=(V, E)$, which means for $v \in V$, they are labeled with $0-|V|-1$ randomly and we only consider iterating from vertices with small labels to large labels.

We define a random path $\{v_0, v_1, v_2, \ldots, v_{l-1}\}$ of length $l$ in a random graph $G=(V, E)$: $v_i$ is an vertex $v \in V$ and there are no repeated vertices in $\{v_0, v_1, v_2, \ldots, v_{l-1}\}$. Given an iteration method $I \in \{\text{BSP, ASYN, HYBRID}\}$, we define a function $\text{PIL}_I(l)$ to get the expected PIL of a random path of length $l$, and function $\text{PIL}_I(l, v)$ to get the expected PIL of a random path of length $l$ ending with vertex $v$ where $0 \leq v < |V|$.

For BSP, a vertex can only access other vertices’ values from last iteration, so we always have:

$$\text{PIL}_{\text{BSP}}(l) = l$$

For ASYN, we recursively compute the PIL:

$$\text{PIL}_{\text{ASYN}}(l, v) = \frac{\sum_{i=0}^{v-1} \text{PIL}_{\text{ASYN}}(l-1, i) + \sum_{i=v+1}^{|V|-1} (\text{PIL}_{\text{ASYN}}(l-1, i) + 1)}{|V|-1}$$

$$\text{PIL}_{\text{ASYN}}(l) = \frac{\sum_{v=0}^{|V|-1} \text{PIL}_{\text{ASYN}}(l, v)}{|V|}$$

To derive $\text{PIL}_{\text{ASYN}}(l)$, we use $\text{PIL}_{\text{ASYN}}(l, v)$ as a helper function to get the expected PIL of every random path ending with $v \in V$. Since the probability of $v$ ending between $0-|V|-1$ is the same, so we can just sum them up and divide by $|V|$. We also use $\text{PIL}_{\text{ASYN}}(l-1, v)$ to recursively get $\text{PIL}_{\text{ASYN}}(l, v)$. We divide random paths of length $l-1$ into two cases: those ending with a vertex less than $v$ and those ending with a vertex greater than $v$. In the first case, we do not need an extra iteration to pass a message, because that value is available in the current iteration. For the second case, we do need another iteration to pass the message, so we add one to the PIL. Finally, we sum them up and divide the sum by $|V|-1$. Because the diameters (length of the longest path among all shortest paths between any two connected vertices) of natural graphs is normally very small compared to $|V|$ [48], we ignore that we may encounter some vertices already in the path. Otherwise, we need to enumerate all paths with length $l$ in the graph, and the complexity would be close to $|V|^l$, which is not computable even for a small graph.

For the HYBRID model, we define another function $\text{PIL}_{\text{HYBRID}}(l, p)$ to return the PIL of a random path of length $l$ ending with a vertex in partition $p$. We assume we have $np$ total partitions. Thus, we have the following formula:

$$\text{PIL}_{\text{HYBRID}}(l, p) = \frac{\sum_{i=0}^{p-1} \text{PIL}_{\text{HYBRID}}(l-1, i) + \sum_{i=p}^{np-1} (\text{PIL}_{\text{HYBRID}}(l-1, i) + 1)}{np}$$

$$\text{PIL}_{\text{HYBRID}}(l) = \frac{\sum_{p=0}^{np-1} \text{PIL}_{\text{HYBRID}}(l, p)}{np}$$
Similar to $P_{ASYN}(l)$, we use $P_{HYBRID}(l, p)$ as a helper to get the expected PIL of every random path ending in partition $p$ and since the probability of ending between $0–p – 1$ is the same, we sum them up and divide by $np$. We also use $P_{HYBRID}(l – 1, p)$ to recursively get $P_{HYBRID}(l, p)$. We again divide random paths of $l – 1$ into two cases: the first ending in a partition less than $p$ and the second ending in a partition not less than $p$. For the first case, we do not need an extra iteration to pass the message. For the second case, we do need another iteration, so we increment the PIL by 1 here. Finally, we sum them up and divide by $np$. Because the graph diameter is normally much smaller than $|V|$, we again ignore that we may encounter some vertices already in the path.

Table 23: Expected PIL for the 3 Models

<table>
<thead>
<tr>
<th>Path Length</th>
<th>BSP</th>
<th>ASYN</th>
<th>HYBRID-128</th>
<th>HYBRID-32</th>
<th>HYBRID-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.0</td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2.0</td>
<td>2.0</td>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4.0</td>
<td>4.0</td>
<td>4.3</td>
<td>5.1</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>8.0</td>
<td>8.1</td>
<td>8.5</td>
<td>10.3</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>16.0</td>
<td>16.3</td>
<td>17.0</td>
<td>20.6</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>32.0</td>
<td>32.5</td>
<td>34.1</td>
<td>41.1</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>64.0</td>
<td>65.0</td>
<td>68.1</td>
<td>82.3</td>
</tr>
</tbody>
</table>

Table 23 shows the PILs for the 3 different execution models with a random graph of 10M vertices. The HYBRID models are parameterized by the number of partition: $HYBRID – n$ means there are $n$ partitions. Because a natural graph normally does not have a diameter larger than 50 [48], we only simulate up to 128 here. Table 23 shows that the BSP model is the slowest to converge and ASYN’s speed is about twice that of BSP for a random graph. When the HYBRID model has 128 or 32 partitions, it is very close to the convergence speed of ASYN. When the HYBRID model uses 8 partitions, the convergence speed is about 30% slower than ASYN. Please note that for a not very small graph, GraphStone could easily execute it with more than 32 partitions without noticeable performance degrade per iteration. And even for a very small graph, GraphStone would not be slower than BSP model and only the performance gain would be smaller.
References


