THE UNIVERSITY OF CHICAGO

NEW METHODS FOR GRAPH COMPUTATION ON SINGLE NODES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES DIVISION
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
ZHIXUAN ZHOU

CHICAGO, ILLINOIS
SPRING, 2018
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... vi

LIST OF TABLES ........................................................ vii

ACKNOWLEDGMENTS ...................................................... viii

ABSTRACT ................................................................. ix

1 INTRODUCTION ......................................................... 1

2 RELATED WORK ......................................................... 7
  2.1 Process Out-of-core Graphs ........................................ 7
  2.2 Process In-memory Graphs ......................................... 8
  2.3 Graph Programming Systems’ Benefits ............................ 8
  2.4 Major Programming Models ....................................... 10
  2.5 Tension Between Parallelism and Iteration Count .............. 11

3 GRAPHZ: DEGREE-ORDERED STORAGE AND ORDERED DYNAMIC MESS- 
AGES ................................................................. 14
  3.1 Degree-ordered Storage ........................................... 14
  3.1.1 The Vertex Index ................................................ 14
  3.1.2 Example ......................................................... 15
  3.1.3 Implementation ................................................ 17
  3.1.4 Analysis of Unique Degrees .................................. 18
  3.1.5 Analysis of Edge Density Distribution ....................... 19
  3.2 Programming in GraphZ ........................................... 21
  3.2.1 Writing GraphZ Programs ..................................... 21
  3.2.2 Execution Model ............................................... 22
  3.2.3 Ordering Guarantees .......................................... 24
  3.2.4 PageRank Example ............................................ 25
  3.2.5 GraphZ’s Expressiveness ..................................... 26
  3.2.6 Ease of Use ..................................................... 27
  3.2.7 Ordered Dynamic Message Model ............................. 28
  3.3 Implementation ................................................... 29
  3.3.1 Sio & Dispatcher ............................................... 30
  3.3.2 Worker & Dynamic Messages .................................. 31
  3.3.3 MsgManager & Dynamic Messages ........................... 32
  3.4 Empirical Evaluation ............................................. 33
  3.4.1 Experimental Setup ........................................... 33
  3.4.2 Preprocessing And Vertex Index Size ....................... 35
  3.4.3 Extra Large Graph Performance ............................. 36
  3.4.4 Large Graph Performance .................................... 36
LIST OF FIGURES

2.1 Simple example of how graph structure effects runtime for bulk and asynchronous execution models. ................................. 11
3.1 Example of Relabeling and tight ID slots ............................................. 15
3.2 CDF of in-partition messages as a function of partition size. .................... 20
3.3 Graph Processing in the View of Developers ........................................ 23
3.4 GraphZ Implementation Overview ....................................................... 29
3.5 Run time comparison on the xlarge graph. ........................................... 35
3.6 Run times for different graph sizes ..................................................... 41
3.7 Performance breakdown for the large graph. ....................................... 42
3.8 Total IO volume for large graph ......................................................... 42
4.1 GraphStone Overview. ........................................................................ 43
4.2 Value Exposing View ......................................................................... 48
4.3 GraphStone Implementation Overview ............................................... 49
4.4 Switch between In-memory and Out-of-core Processing ......................... 52
4.5 Summary data showing GraphStone’s speedups over best in class frameworks for In-Memory (a) and Out-of-Core (b) processing. .......... 55
4.6 Run times for graphs processed in memory. ........................................ 57
4.7 Memory Bandwidth (GB/s) for different graphs. .................................... 59
4.8 Run times for graphs processed out-of-core. ....................................... 60
4.9 Amount of I/O used comparisons. ...................................................... 62
4.10 I/O bandwidth comparisons. ........................................................... 63
4.11 Ligra concurrency hazard .................................................................. 67
4.12 Coarse barriers in GraphStone .......................................................... 67
4.13 Run times breakdown for graphs processed in memory. ....................... 73
4.14 R-square between speedups and I/O ratios ....................................... 74
4.15 Iteration number comparisons. .......................................................... 75
# LIST OF TABLES

2.1 Lines of Code to Implement Page Rank. ........................................ 10  
2.2 Time to Execute Page Rank. ................................................... 10  
2.3 Iteration Number Comparison ................................................. 11  

3.1 Example Graph ................................................................. 15  
3.2 Relabeling of Example Graph ................................................. 16  
3.3 Edge List Stored on External Disk .......................................... 16  
3.4 ids_table ................................................................. 17  
3.5 id_offset_table ............................................................. 17  
3.6 SNAP graph properties. ....................................................... 19  
3.7 LOC Comparison of Graph Engines. ......................................... 28  
3.8 Graph Properties. ............................................................ 34  
3.9 Vertex index size executing PageRank. ..................................... 34  
3.10 Preprocessing time (s). ....................................................... 35  
3.11 Iterations for Convergence .................................................. 39  

4.1 LOC Counts for Different Graph Frameworks. .............................. 47  
4.2 Fast Shuffling for Graphs Vertices ........................................ 53  
4.3 Iteration Number after Random Shuffling .................................. 53  
4.4 Graph Properties. ............................................................ 55  
4.5 Run times (s) for X-Stream. .................................................. 58  
4.6 Run times (s) for hand-tuned C code. ...................................... 58  
4.7 Iteration Number for twitter ................................................ 64  
4.8 Iteration Number for lj ..................................................... 65  
4.9 Compare Major features between X-Stream and GraphStone .......... 65  
4.10 Compare iteration numbers and activated vertices for GraphStone over Ligra . 66  
4.11 Major performance factors breakdown ..................................... 69  
4.12 Estimate I/O ratios between GraphStone and X-Stream by iteration number and activate edges ................................................. 71  

A.1 Expected PIL for the 3 Models ............................................. 81
ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my advisor, Prof. Hank, for his valuable guidance throughout this work. This work could not have been accomplished without his experienced target control and patient help on all steps. Also, I am grateful to Prof. Andrew to bring this prosperous topic out, so that I got such a chance to do something on graph computing.

I would thank all my friends who give suggestions during this work. Finally, I would thank my family for their selfless and forever love.
ABSTRACT

There has been a recent explosion in specialized graph computing frameworks with different classes of framework addressing different niches of requirements. Specifically, some are designed for graphs that fit in a single machine’s memory, but fail when the graph exceeds memory size. Other frameworks handle extremely large graphs that exceed a single machine’s memory, but achieve worse performance than simple C programs on small graphs. Thus, the current state-of-the-art requires users to adopt one programming framework for small graphs and a completely different framework for large graphs. In this thesis, we present GraphZ and GraphStone.

GraphZ has two innovations that improve the performance of software frameworks for out-of-core graph analytics. The first is degree-ordered storage, a new storage format that dramatically lowers book-keeping overhead when graphs are larger than memory. The second innovation replaces existing static messages with novel ordered dynamic messages which update their destination immediately, reducing both the memory required for intermediate storage and IO pressure. We implement these innovations in a framework called GraphZ—which we release as open source—and we compare its performance to two state-of-the-art out-of-core graph frameworks. For graphs that exceed memory size, GraphZ’s harmonic mean performance improvements are 1.8–8.3× over existing state-of-the-art solutions. In addition, GraphZ’s reduced IO greatly reduces power consumption, resulting in tremendous energy savings.

GraphStone is a unified framework that outperforms current best-in-class approaches for both in-memory and out-of-core graph computing. GraphStone’s key contribution is an execution model which combines the best merits of bulk synchronous and asynchronous updates to achieve the parallelism and streaming data access of bulk synchronous models with the fast convergence time of asynchronous models. We implement GraphStone in C++ and compare its performance to best-in-class approaches for both in-memory and
out-of-core computing. We find that GraphStone programs require no modification when moving from small to large graphs, yet GraphStone is $2.5 \times$ faster than a state-of-the-art in-memory approach and $11.6 \times$ faster than a popular, robust out-of-core approach.
CHAPTER 1
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. \textit{in-memory} systems designed with the assumption that main memory is large enough to hold the entire graph [36, 38, 45, 46] and

2. \textit{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 [25]. 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, using 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× an 13× compared to GraphChi and X-Stream, respectively.
• For a graph that exceeds SSD capacity, GraphChi fails, but GraphZ’s harmonic mean speedup is $1.8 \times$ 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× faster than Ligra. It is also over 7× faster than a hand-tuned serial C implementation and 36× faster than X-Stream, which is optimized for out-of-core.

- For out-of-core graph computations, GraphStone is 11.6× than X-Stream.
CHAPTER 2

RELATED WORK

2.1 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. Other approaches to IO reduction for out-of-core graph analytics include PathGraph [63], GraphQ [54], and MMap [31].

2.2 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.

2.3 Graph Programming Systems’ Benefits

McSherry et al. question the need for any graph programming system [36]. They compare many of the above frameworks to simple PageRank 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.

To address this study, we reproduce some of the results here. Specifically, we compare the LOC and run time required for PageRank written in C, GraphChi, and GraphZ for both in-core and out-of-core graphs. Our test setup is detailed in Sec. 3.4. In brief, we use the LiveJournal graph [61] to test in-memory performance and YahooWeb graph [59] to test out-of-core performance.

Table 2.1 shows the LOC required to implement PageRank, while Table 2.2 shows the runtime. We conclude:

- Programming systems provide no benefit when graphs fit in memory – the C code is almost three times faster. The slowdown makes sense as any out-of-core framework adds book-keeping overhead that is unnecessary when running in-core. This observation is consistent with [36].

- Frameworks provide a huge benefit for graphs that must be processed out-of-core, including both code size reductions and time savings because the programming systems’ runtimes automatically overlap IO and computation. GraphZ’s innovations provide further execution time reduction over GraphChi.

These results confirm the need for graph programming systems supporting out-of-core processing. With companies like Facebook claiming their internal graphs with trillions of edges [7], we believe the need for out-of-core frameworks will only become greater.
Table 2.1: Lines of Code to Implement Page Rank.

<table>
<thead>
<tr>
<th>Graph Size</th>
<th>C</th>
<th>GraphChi</th>
<th>GraphZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>in memory</td>
<td>300</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>out-of-core</td>
<td>500</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 2.2: Time to Execute Page Rank.

<table>
<thead>
<tr>
<th>Graph Size</th>
<th>C</th>
<th>GraphChi</th>
<th>GraphZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>in memory</td>
<td>6s</td>
<td>22s</td>
<td>16s</td>
</tr>
<tr>
<td>out-of-core</td>
<td>3848s</td>
<td>2958.2</td>
<td>2024.3s</td>
</tr>
</tbody>
</table>

2.4 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 a 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 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.
2.5 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. 2.1. We always iterate from left to right (from vertex 0 to 6), as all frameworks iterate through vertices in numerical order.

Table 2.3 compares the iteration counts for BSP and ASYN in the best and worst case. Table 2.3(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 2.3(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 in this iteration. Vertices 0–4, 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.
CHAPTER 3
GRAPHZ: DEGREE-ORDERED STORAGE AND ORDERED
DYNAMIC MESSAGES

3.1 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.

3.1.1 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
3.1.2 Example

We illustrate degree-ordered storage using the graph in Figure 3.1a. Table 3.1 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 3.2 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 3.1b shows the relabeled graph. Having relabeled the graph, we store: the map between the old and new IDs (columns 1 &
4 from Table 3.2), the ordering on the adjacency lists (Table 3.3), 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 3.4 is the lookup table mapping degree to the first ID having this degree, called the 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 3.5 is the lookup table mapping degree number to the offset of the first id having this degree. We call it the id_offset_table. Combined with the 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 ids_table and 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 x, we do a binary search on ids_table to find the degree d satisfying ids_table[d] <= x < ids_table[d + 1]. This d is the out-degree of x. The first id that has
out-degree of $d$ is $\text{ids\_table}[d]$. Then look in the $\text{id\_offset\_table}$ to get the offset of vertex $\text{ids\_table}[d]$. Finally we compute the offset of vertex $x$, by the formula:

$$\text{offset} = \text{id\_offset\_table}[d] + (x - \text{ids\_table}[d]) \times d$$

(3.1)

For example, to find the offset of vertex 3, we do a binary search on $\text{ids\_table}$ to find the degree of vertex 3 is 2 and the first vertex with out-degree 2 is 2. Then we check the $\text{id\_offset\_table}[2]$ 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.

### 3.1.3 Implementation

The conversion to DOS requires only sequential access to external storage. We convert the graph to a new list, $\text{EDGES}$, of triad $\langle \text{src, dest, deg} \rangle$, where $\text{deg}$ is the degree of $\text{src}$. Then we use external k-way merge sort [12, 22, 49] to sort it using $\text{deg}$ as 1st key and $\text{src}$ as 2nd key. Thus, we can relabel all those $\text{srcs}$, with sequential access. At the same time we record the mapping from $\text{newid}$ to $\text{oldid}$. After this, we get a list of pair $\langle \text{newid, oldid} \rangle$, then we do another sort by key $\text{oldid}$, and we record this sequential mapping from $\text{oldid}$ to $\text{newid}$ on external storage.

Next, we sort $\text{EDGES}$ by $\text{dest}$. Thus, with the mapping from $\text{oldid}$ to $\text{newid}$, we sequentially relabel $\text{dests}$ in $\text{EDGES}$. Also, we need to fill in those vertices with 0 degrees at the same time. After this, we have finished the relabeling for all vertices. At
last, we sort \( \text{EDGES} \) by \( \text{src} \) again, and generate all those support files for each partitions. \( \text{deg} \) in \( \text{EDGES} \) is not required after the first sorting, so that we can reduce disk accesses.

After relabeling and building the indexing data structures, they can then be used for many different computations; \( \text{i.e.} \), 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. 3.4.2).

### 3.1.4 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 \( V_2 = \{v \in V | \text{degree}(v) \geq \sqrt{|E|} \} \).

Function \( \text{degree}(v) \) returns the degree of vertex \( v \).

Assume \( |UD_2| > \sqrt{|E|} \), then

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

\[
= |V_2| \times \sqrt{|E|} \geq |UD_2| \times \sqrt{|E|}
\]

\[
> \sqrt{|E|} \times \sqrt{|E|} = |E|
\]

Thus we get the contradiction: \( |E| > |E| \), so the assumption is wrong. And we have
Table 3.6: 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>

\[ |UD_2| \leq \sqrt{|E|}. \]

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 3.6 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.

3.1.5 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 have highest degrees, compared to other partitions.

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

Fig. 3.2 shows the ratio of edges within the top n% of vertices for natural graphs used in our evaluation (see Sec. 3.4.1). 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.
3.2 Programming in GraphZ

GraphZ inherits the vertex-centric programming model common to many graph programming systems and augments it with novel ordered dynamic messages. These messages combine data and computation, eliminating much of the IO required for prior approaches’ static messages. Additionally, message ordering means that – despite their dynamic nature – all vertices complete message updates in the same sequence, making it easier to debug. We describe how users write GraphZ programs. Where appropriate, we compare to GraphChi [25] and X-stream [42], existing open-source, out-of-core graph programming systems.

3.2.1 Writing GraphZ Programs

Following the vertex-centric model, GraphZ users specify a VertexDataType and an update() function. Additionally, GraphZ users specify a MessageDataType and an apply_message() function. The GraphZ runtime iterates over the vertices, calling update() at each and intercepting any messages to determine whether the destination is in memory or on disk. If in memory, the runtime calls apply_message() directly on the destination. If on disk, the runtime stores the message data and calls apply_message() when the destination is loaded, preserving ordering.

User-defined Datatypes As shown in Algorithm 10, GraphZ users define VertexDataType and MessageDataType as structs of existing C++ types, including other structs. For example, to implement the PageRank algorithm, we define MessageDataType as float and VertexDataType as a struct of two floats – one for storing the rank value and another one for accumulating messages. In breadth-first-search, users need only define the MessageDataType as int and the VertexDataType as a struct of two ints, one for current label and another for a possible value change.
Algorithm 1 Key Data Structures

```plaintext
1: struct VertexDataType {
2: int vval1;
3: float vval2;
4: double vval3;
5: ..... ▷ it could be int, float, struct, array....
6: }
7: }
8: struct MessageDataType {
9: int msg_val1;
10: float msg_val2;
11: double msg_val3;
12: ..... ▷ it could also be int, float, struct, array....
13: }
```

User-defined Functions  GraphZ users define `update()` and `apply_message()`. As Algorithm 2 shows, `update()`: (1) adjusts a vertex’s value if needed, (2) computes new messages, (3) iterates over out-edges, possibly sending messages. The `apply_message()` routine defines the computation associated with each message. In Algorithm 2, `apply_message()` calls the function `f_2` and returns the new vertex data of the destination vertex.

Algorithm 2 User-defined methods in GraphZ

```plaintext
1: function UPDATE(vertex)  ▷ not a must
2: vertex ← f_1(vertex)
3: for vadj in vertex’s adjacent vertices do
4: if some condition then
5: compute a message msg
6: send the message msg to vadj
7: function APPLY_MESSAGE(vertex, msg)
8: vertex ← f_2(vertex, msg)
9: return vertex
10: ▷ f_2 is often a very simple function, like min(vertex, msg), vertex + msg, vertex.append(msg)
```

3.2.2 Execution Model

GraphZ applies the `update()` methods using the asynchronous execution model shown in Figure 3.3.

GraphZ’s runtime manages two lists: `vertices_array` and `vertices_adjacents_list`. `vertices_array` stores all vertices’ values and `vertices_adjacents_list` stores every
Iterate one by one

Figure 3.3: Graph Processing in the View of Developers
vertex’s adjacency list in degree-sorted order (see Sec. 3.1). GraphZ’s runtime iterates through vertices_array, calling update() on every vertex. Fig. 3.3, shows arrows pointing two different directions. The upward arrows indicate that when GraphZ updates a vertex, it may (depending on the update() method) read that vertex’s value and combine it with the adjacency list to adjust this value. Downward arrows represent sending messages to a vertex’s out-neighbors. The runtime automatically schedules the execution of apply_message() every time a message is sent.

Like GraphChi and X-Stream, GraphZ is inherently iterative, and it allows users to choose one of two methods for termination. First, if all values a user cares about are no longer changing, or only changing slightly, users can end the iteration. Second, the developer can specify an exact iteration number, and GraphZ will stop when it reaches such a number.

3.2.3 Ordering Guarantees

Though GraphZ is a multi-threaded graph engine, it provides a strong order consistency guarantee, so users can reason about GraphZ programs as if they were executed by a sequential program. Specifically, GraphZ orders all vertices. If o(v) is a function returning a unique integer representing vertex v’s place in the order, then for any 2 vertices v1 and v2, if o(v1) < o(v2), then the update() method will be called on v1 first in every iteration and messages sent during the update execution of v1 are also applied before updating v2. Besides ease-of-use, maintaining consistency also has a performance advantage, which can greatly accelerate the convergence speed and reduce disk accesses [25]. To be clear, the user does not get to specify the order, which is determined by GraphZ’s degree-ordered storage format (see Sec. 3.1). GraphZ does, however, guarantee that once that order has been determined, it will be the same for each invocation of a particular algorithm and graph.
Algorithm 3 PageRank Data Structures

1: struct VertexDataType {
2:     float vval = 1;
3:     float votes = 0;
4: }
5: struct MessageDataType {
6:     float msg;
7: }

Algorithm 4 PageRank

1: function Update(vertex)
2:     ndeg ← number of vertex’s adjacent vertices
3:     vertex.vval ← 0.15 + 0.85 * vertex.votes
4:     vertex.votes ← 0
5:     if ndeg == 0 then return
6:     if cur_iter == 0 then
7:         msg ← 1/ndeg
8:     else
9:         msg ← vertex.vval/ndeg
10:    for vadj in vertex’s adjacent vertices do
11:        send message vertex.vval to vadj
12: function apply_message (vertex, msg)
13:    vertex.votes ← vertex.votes + msg
14: return vertex

3.2.4 PageRank Example

We illustrate PageRank in GraphZ. On initialization, we assign all vertices rank 1 and received votes to 0. In Equation 3.2[39], vertices B, C, D and etc. are

\[
PR(A) = 1 - d + d \left( \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \ldots \right)
\]  \tag{3.2}

First, as shown in Algorithm 3, we define VertexDataType as a struct of two floats (vval and new_vval) and we define MessageDataType as float. We define two values for the vertex tracking its (1) rank and (2) received votes.

As Algorithm 4 shows, Step 3 in update() computes the vertex value according to Equation 3.2, then computes the vote to send to its out-neighbors. In apply_message(), it adds the msg to the target vertex to be computed in future update().
### 3.2.5 GraphZ’s Expressiveness

The GraphZ interface shares features of GraphChi, but it requires users to define the `apply_message()` method. These changes greatly reduce GraphZ’s storage requirements and accesses to the backing store compared to GraphChi. We argue that these changes do not reduce expressiveness compared to GraphChi.

We demonstrate this claim by showing how to convert a GraphChi program into a GraphZ program. As shown in Algorithm 5, we first define a new structure, `Edge`, to represent the key structure `Edge` in GraphChi. Under `Edge`, `neighbor` is the id of a vertex’s neighbor and `edge_val` corresponds to the edge value in GraphChi. We add `edges` to `VertexDataType`, thus a vertex’s edges are part of the vertex. `RealVertexDataType` is the value stored at the vertex.

**Algorithm 5** Data Structures for Emulating GraphChi  

```cpp
1:  struct Edge {
2:    vertex_id neighbor;
3:    EdgeDataType edge_val;
4:  }
5:  struct VertexDataType {
6:    List <Edge> edges;
7:    RealVertexDataType vertex_val;
8:  }
9:  struct MessageDataType {
10:   Edge edge;
11:  }
```

Algorithm 6 makes the `update()` function compatible with GraphChi. The variable `edges` is treated exactly as in-edges in GraphChi as they can be read and used to update the real `vertex_val`. Then, the algorithm does the same as GraphChi, iterating over out-edges and sending messages. The `apply_message()` function performs no operations, but adds `edge` to the vertex’s `edges`. This copies the process of writing messages to out-edges in GraphChi. Through this process, any GraphChi program can be converted to a GraphZ program. Thus, we conclude that GraphZ is as least as expressive as GraphChi. This
construction does not take advantage of GraphZ’s dynamic messages, but it shows that even graph algorithms that do not perform commutative and associative operations on inbound edges can be implemented in GraphZ.

Algorithm 6 Covert GraphChi Programs to GraphZ

1: function Update(vertex)
2:     for edge in vertex’s edges do
3:         read edge data from edge
4:         if some condition then
5:             update vertex_val
6:     ▷ this self update could also be done outside this for loop
7:     for vadj in vertex’s adjacent vertices do
8:         Construct a msg
9:         msg.edge.neighbor ← vertex’s id
10:        if some condition then
11:           compute a message real_msg
12:           msg.edge.edge_val ← real_msg
13:           send the message msg to vadj
14:     vertex.edges ← ∅
15: function apply_message (vertex, msg)
16:     vertex.edges.append(msg.edge) return vertex

3.2.6 Ease of Use

GraphZ is at least as expressive as GraphChi, but we need to evaluate if defining the apply_message() function adds significant burden to the programmer. We therefore compare the LOC (lines of code) required to write our test algorithms in GraphZ, GraphChi, and X-Stream.

Table 3.7 shows the LOC needed for the six algorithms used in our experimental evaluation (see Sec. 3.4.1). These numbers show that both GraphChi and GraphZ require similar LOC counts; X-Stream is uniformly higher. We conclude that GraphZ does not add additional burden to programmers familiar with GraphChi and results in significantly simpler code than X-Stream.
Table 3.7: LOC Comparison of Graph Engines.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GraphChi</th>
<th>X-Stream</th>
<th>GraphZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>34</td>
<td>99</td>
<td>25</td>
</tr>
<tr>
<td>CC</td>
<td>32</td>
<td>64</td>
<td>13</td>
</tr>
<tr>
<td>PR</td>
<td>23</td>
<td>60</td>
<td>24</td>
</tr>
<tr>
<td>BP</td>
<td>30</td>
<td>254</td>
<td>50</td>
</tr>
<tr>
<td>RW</td>
<td>30</td>
<td>65</td>
<td>30</td>
</tr>
<tr>
<td>SSSP</td>
<td>32</td>
<td>59</td>
<td>30</td>
</tr>
</tbody>
</table>

3.2.7 Ordered Dynamic Message Model

GraphZ supports ordered dynamic messages (DM) while maintaining the efficiency of GraphChi’s asynchronous model. Despite the name, ordering is important in the asynchronous model as a strict ordering allows programmers to reason about the code (making it possible to debug) and provides deterministic performance (so each execution always performs the same sequence of operations). Internally, GraphZ uses Worker and MsgManager modules to implement DM. GraphZ’s message intercepting mechanism ensures this execution order. When an `update()` method sends a message, a Worker intercepts it and determines if this message should be applied immediately (because the destination vertex is in memory) or forwarded to the MsgManager (because the destination is on disk).

This interception process is transparent to developers while ensuring the order in Sec. 3.2.2. the Worker iterates over vertices applying the `update()` function defined by developers. When executing `update()`, it will call the `apply_message()` function. `apply_message()` gets the destination vertex ID and determines its partition. If the destination vertex belongs to current active partition, the message will be applied immediately. If the destination vertex is in another partition (currently on disk), the message will be forwarded to MsgManager and be appended to that partition’s buffer waiting to be written to disk.

By intercepting messages, a message whose destination vertex is in the current partition has the message applied immediately. In that way the GraphZ runtime enforces the
rule that a vertex’s out-messages are always applied before a vertex with larger ID enters `update()` within current active partition. For messages that go to other partitions, the MsgManager ensures the order.

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 GraphZ runtime retrieves edges from storage and the process begins on the loaded partition. In this way, the GraphZ runtime can automatically overlap communication and computation and provide greater performance than that achievable with straightforward C implementations (see Sec. 2.3).

### 3.3 Implementation

We describe GraphZ’s implementation with a focus on the support for ordered dynamic messages.

![GraphZ Implementation Overview](image.png)

**Figure 3.4: GraphZ Implementation Overview**

Fig. 3.4 shows the four major components of GraphZ’s runtime: *Sio*(Sequential I/O), *Dispatcher*, *Worker*, and *MsgManager*. The runtime divides a large graph into partitions...
which fit into available memory. To iterate over a partition, the MsgManager loads a partition’s vertices into memory then calls \texttt{apply\_message()} on any destination vertices within the partition, enforcing the message ordering. Next, Sio reads the graph storage file into memory and reads \textit{edge blocks} that are passed to the Dispatcher, which translates them into adjacency lists. The Worker calls \texttt{update()} on each vertex in the partition. The Worker also checks each message’s destination and calls \texttt{apply\_message()} on any recipient vertices in the current partition. All other messages are passed to MsgManager, which writes them to disk. The Worker and MsgManager combine to implement the GraphZ messaging model.

Each component is implemented with a separate thread pool allowing the runtime to take advantage of multicore resources when applicable. The four components are connected by lock-based circular queues. If one of them fails to insert or get a task from the queue, it will be blocked and put to sleep. Sleeping the threads results in significant power savings during times of heavy I/O. Our attempts to implement lock-free queues resulted in worse performance, so we use traditional locking schemes. All data transferred between the four main components are organized in small bundles, which enables the runtime to start updating vertices even if only a small part of the current active partition has been loaded. Thus, for large partitions, GraphZ achieves high overlap between computation and I/O.

\subsection*{3.3.1 Sio \& Dispatcher}
Sio is short for Sequential I/O retriever. At the start of every partition, Sio loads graph data into memory. GraphZ’s degree-ordered storage makes it possible to load the entire vertex index into memory. Vertices within a partition are always read in order, taking advantage of system-level prefetching and caching.

To maximize IO bandwidth and get better pipelining, the Dispatcher constructs ad-
jacency lists from file data. The Dispatcher receives blocks from Sio and parses the block into several adjacency lists. For example, Sio tells the Dispatcher the length $len$ of a block, the startId $l$, and the endId $r$. Then the Dispatcher knows that there are $r - l$ adjacency lists in the block and every vertex has $\frac{len}{r - l}/\text{sizeof}(\text{Vertex\_ID})$ edges. The Dispatcher allocates memory for each vertex’s adjacency list.

### 3.3.2 Worker & Dynamic Messages

The Worker performs two tasks: 1) iterating over vertices and 2) intercepting messages to in-memory vertices.

GraphZ adds support for dynamic messages, while maintaining the efficiency of asynchronous execution. Ordering allows programmers to reason about the code – enabling debugging – and provides deterministic performance – because each execution performs the same sequence of operations. GraphZ’s message intercepting mechanism ensures this execution order.

**Algorithm 7** Message Intercepting (Part a.)

```plaintext
1: procedure Main Worker
2:   for vertex in current partition do
3:     do update
4: 
5: function SEND_MESSAGE(msg, vertex)
6:   if vertex belongs to current partition then
7:     execute apply_message on vertex with msg
8:   else
9:     forward vertex and msg to the MsgManager
```

This interception process is transparent to developers while ensuring the order in Sec. 3.2.2. As shown in Algorithm 7, the Worker iterates over vertices applying the `update()` function. When executing `update()`, it will call the `apply_message()` function. `apply_message()` gets the destination vertex ID and determines its partition. If the destination vertex belongs to current active partition, the message will be applied imme-
diately. If the destination vertex is in another partition (currently on disk), the message will be forwarded to MsgManager and be appended to that partition’s buffer waiting to be written to disk.

By intercepting messages, a message whose destination vertex is in the current partition has the message applied immediately. In that way the GraphZ runtime enforces the rule that a vertex’s out-messages are always applied before a vertex with larger ID enters update() within current active partition. For messages that go to other partitions, the MsgManager ensures the order.

### 3.3.3 MsgManager & Dynamic Messages

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, see Algorithm 8.

![Algorithm 8 Message Intercepting (Part b.)](image)

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.

3.4 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.

3.4.1 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.

**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, PageR-
Table 3.8: Graph Properties.
(M = Million, B = Billion, GB = Gigabytes.)

<table>
<thead>
<tr>
<th>Graph</th>
<th>LiveJournal</th>
<th>Friendster</th>
<th>YahooWeb</th>
<th>Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>small</td>
<td>medium</td>
<td>large</td>
<td>xlarge</td>
</tr>
<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 3.9: 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>

ank, 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 4.4 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.
Table 3.10: Preprocessing time (s).

<table>
<thead>
<tr>
<th>Graphs</th>
<th>small HDD</th>
<th>small SSD</th>
<th>medium HDD</th>
<th>medium SSD</th>
<th>large HDD</th>
<th>large 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>

3.4.2 Preprocessing And Vertex Index Size

To demonstrate degree ordered storage’s benefits, Table 3.9 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 3.10 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.

Figure 3.5: Run time comparison on the xlarge graph.
3.4.3 Extra Large Graph Performance

The results for the extra large graph (Sim) on HDD are shown in Fig. 3.5 (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 3.9). The harmonic mean of speedup shows GraphZ is $1.86 \times$ faster than X-Stream. GraphZ’s maximum speedup is $3.06 \times$ 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.

3.4.4 Large Graph Performance

The results for the large graph (YahooWeb) are shown in Fig. 3.6(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 $4.84 \times$ faster than GraphChi and $3 \times$ faster than X-Stream. GraphZ’s maximum speedup is $8 \times$ compared to GraphChi on SSSP and $7.5 \times$ compared to X-Stream on RW. For the SSD, the harmonic mean of speedup shows GraphZ is $1.80 \times$ faster than GraphChi and $1.85 \times$ faster than X-Stream. The maximum speedup of GraphZ is $3.7 \times$ compared to GraphChi on BFS and $13 \times$ compared to X-Stream on RW. All approaches benefit tremendously from moving to SSD. GraphZ still provides a significant performance gain,
3.4.5 Medium Graph Performance

Fig. 3.6(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.

3.4.6 Small Graph Performance

The results for the small graph are shown in Fig. 3.6(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. 3.6(b), however, shows the results of in-memory graph processing with GraphZ are competitive or even sometimes much better than existing approaches.
3.4.7 Performance Breakdown

We analyze each of the proposed technique’s contributions to performance in Fig. 3.7. 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.

3.4.8 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 3.11: 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>GraphChi</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>X-Stream</td>
<td>47</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>GraphZ</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Medium</td>
<td>GraphChi</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>X-Stream</td>
<td>59</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>GraphZ</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.11 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 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.

3.4.9 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. 3.8 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.
Figure 3.6: Run times for different graph sizes.
Figure 3.7: Performance breakdown for the large graph.

Figure 3.8: Total IO volume for large graph
CHAPTER 4

GRAPHSTONE: HYBRID MODEL AND ON THE FLY VERTEX ASSEMBLY

4.1 GraphStone Design and Implementation

GraphStone adopts the popular vertex-centric programming model. Fig. 4.1 illustrates this model and GraphStone’s unique twist. Like existing frameworks, GraphStone iterates through the vertex array calling a user-defined `update()` method on each. Unlike existing in-memory frameworks, GraphStone divides the vertices into partitions. All updates to active vertices (white in the figure) within a partition are done in parallel; i.e., the updates within a partition are bulk synchronous. An update might activate new vertices. The distinction between activated and deactivated vertices saves time and memory resources as the GraphStone runtime only constructs full vertices for active vertices. The `update()` method sees the most recent vertex values for all earlier partitions; i.e., the updates between partitions are asynchronous.

This hybrid synchronization model is one contribution of GraphStone. Another is a small change to the interface, which allows it to be implemented efficiently whether data is in memory or on disk. While most prior approaches allow `update()` methods to directly generate messages (whose values must be stored), GraphStone requires users to specify a `gen_msg()` method that generates a message when called. This additional method allows the runtime to determine when message data is available and avoid unnecessary writes to either memory or disk. For example, if a user wants to send a message to a vertex
Algorithm 9 GraphStone Runtime Execution

1: User-specified functions are denoted in blue
2: function start
3: initialize vertices array with vertex_constructors
4: mark each vertex as activated or deactivated
5: iterate
6: function iterate
7: iter ← 0
8: while iter ≤ max_iter and need_more_iteration do
9:   need_more_iteration ← false
10:  iterate_once(iter)
11: function iterate_once(iter)
12:   for p ∈ Partitions do
13:      parallel for v ∈ p
14:      if v is activated then
15:         Construct message iterator it with v’s incoming messages
16:         ▷ it implicitly calls gen_msg()
17:         UPDATE(v,it,iter)
18:      mark v as deactivated

on disk, prior approaches store the data for that message immediately, consuming extra memory [25]. GraphStone instead waits until the vertex is in memory and then generates the message. Since users do not know exactly where messages are, or if they have been constructed yet, GraphStone provides a msg_iterator data type which provides users access to inbound messages during an update.

4.1.1 Writing GraphStone Programs

GraphStone users should understand how the runtime interacts with user code. So, we first detail the runtime’s execution, then describe the user-defined data structures and finally the user-defined functions that will be called by the runtime.

User’s View of GraphStone Execution Algorithm 9 shows GraphStone’s execution. start_engine() is the main GraphStone function. It initializes the vertex array with user-specified vertex constructors. Next, GraphStone initializes the scheduler, marking vertices as activated or deactivated. There are two ways users initialize the scheduler:
(1) mark all vertices active or (2) select a few to be active. For example, in a single-source shortest paths algorithm, only the start vertex will be active initially. In PageRank, all vertices will be initially active. Next, GraphStone iterates over vertices. For each iteration, GraphStone loops over all partitions. Within a partition all activated vertices are updated. After the call to \texttt{update()}, GraphStone marks the vertex as deactivated. Within the \texttt{update()} method, a vertex can activate its neighbors. While this activation process may seem to be subject to race conditions, GraphStone does not have any false-activations or lost activations (the details are covered in the implementation section).

**User-defined Datatypes** GraphStone users define 2 data types: \texttt{vertex\_t} and \texttt{message\_t}. \texttt{vertex\_t} stores each vertex’s value. It contains a constructor to initialize the vertex, taking its vertex id as input. \texttt{message\_t} communicates values between vertices. The runtime automatically constructs an iterator over received messages and passes this to the \texttt{update()} method. As shown in Algorithm 10, if we want to use GraphStone for PageRank, then we define \texttt{vertex\_val\_t} as a struct with a float and a constructor that initializes its value to 1.0. \texttt{message\_t} also contains a single float. To do breadth-first search, we define a \texttt{vertex\_t} to contain a parent vertex id indicating the vertex’s parent in bfs order, which we initialize to \texttt{MAX\_VERTEX\_ID}, meaning the vertex is not yet found.

**User-defined Functions** GraphStone users define 2 functions: \texttt{gen\_msg()} and \texttt{update()}. \texttt{gen\_msg()} generates a message to an adjacent vertex. Its arguments are a vertex, target vertex’s id, and the current iteration number. The runtime passes the iteration number to \texttt{gen\_msg()}, because sometimes users generate different messages depending on iteration number. The return value \texttt{true/false} indicates there is a message from this to the target vertex. The default \texttt{gen\_msg()} always returns \texttt{false} and users can override this behavior. The \texttt{update()} function receives a vertex (not just an id), the current iteration number, and an iterator for incoming messages. The iterator hides some implementation details from the user allowing a common interface whether the graph is processed in-memory or out-of-core.
Algorithm 10 User-defined Data Structures

```c
1: struct vertex_val_t {
2:     float pr;
3:     vertex_id founded;
4:     vertex_id parent;
5:     .......            // The user could freely choose what data types to use
6: }
7: vertex_t (vertex_id t_id){
8:     pr = 1.0;
9:     founded = MAX_VERTEX_ID;
10:    parent = tv_id;
11:    .......            // Member initiation could be done under constructor
12: }
13: }
14: struct message_t {
15:     float pr;
16:     vertex_id parent;
17:     .......            // The user could freely choose what data types to use
18: }
```

Algorithm 11 shows how we define the 2 GraphStone functions for PageRank. For `gen_msg()`, we simply divide the vertex’s pr value by its out-degree and then set the return value to true to indicate there is a message to send, because we need to test for every vertex’s in-neighbors to see whether there is a messages to pull. If the return value is false, then the ”invalid” message by this call to `gen_msg()` does not appear in the iterator provided in the `update` function. Each time `update()` is called on a vertex, the runtime constructs a `msg_it`, allowing the user to iterate through in-bound messages. Line 6-9 computes the new PageRank value for the current vertex. Line 10 sets the pr value to the new one. Line 11 determines whether the current pr value is within the convergence threshold. If yes, then it simply returns; otherwise, it activates its neighbors and signals the runtime that more iterations are required.

Coding Complexity Comparison Table 4.1 compares the lines of code (LOC) between GraphChi, X-Stream, Ligra and GraphStone for five common benchmarks. GraphChi and GraphStone are vertex-centric. X-Stream is edge-centric, and Ligra has a unique programming model. GraphStone has the lowest LOC count for all algorithms, leading us to conclude that it does not significantly increase the programming burden. We note
Algorithm 11 User-defined Functions

1: function gen_msg(vertex, target_id, cur_iter, msg)
2: msg ← vertex.pr / vertex.deg
3: return true

4: function update(vertex, cur_iter, msg_it)
5: new_pr ← 0
6: for msg ∈ msg_it do
7: new_pr ← new_pr + msg
8: new_pr ← new_pr · 0.85 + 0.15
9: swap(new_pr, vertex.pr)
10: if FABS(new_pr − vertex.pr) < threshold then
11: return
12: for neighbor ∈ vertex’s neighbors do
13: activate(neighbor)
14: need_more_iteration
15: return

Table 4.1: LOC Counts for Different Graph Frameworks.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>GraphChi</th>
<th>X-Stream</th>
<th>Ligra</th>
<th>GraphStone</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>34</td>
<td>99</td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>CC</td>
<td>32</td>
<td>64</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>PR</td>
<td>23</td>
<td>60</td>
<td>52</td>
<td>14</td>
</tr>
<tr>
<td>RW</td>
<td>30</td>
<td>65</td>
<td>50</td>
<td>24</td>
</tr>
<tr>
<td>SSSP</td>
<td>32</td>
<td>59</td>
<td>28</td>
<td>15</td>
</tr>
</tbody>
</table>

that Ligra actually requires two implementations, because the Ligra runtime may switch between sparse and dense traversal (definition in Sec. 4.6.3); however, sparse traversal requires the user to take data races into consideration. GraphStone users are hidden from these details.

4.2 Hybrid Execution Model

We describe GraphStone’s execution model, which combines properties of the BSP and ASYN models. Fig. 4.2 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. 4.2a, the set $VE_{BSP}(v, i)$ for BSP is:

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

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. 4.2b, $v$ 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:

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

Section 2.5 argues that BSP is easier to parallelize than the ASYN model. The arrows shown in Fig. 4.2 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 \textit{partitions} vertices into \(np\) partitions, each with \(W\) vertices. We define \(p(v)\) to get the partition id of vertex \(v\). Set \(VE_{HYBRID}(v, i)\) is then:

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

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.

### 4.3 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.

![GraphStone Implementation Overview](image-url)
Fig. 4.3 and Algorithm 12 illustrate 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.

### 4.4 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. 4.4 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 \texttt{mmaps} 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 \texttt{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 \texttt{update()} on every vertex in the \texttt{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
4.5 Shuffling Graphs Without a Lookup Table

Sec. 2.5 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. 2.1(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. 2.1(a), we choose $n_{row}$ to be 2. Table 4.2a shows that
Table 4.2: Fast Shuffling for Graphs Vertices

<table>
<thead>
<tr>
<th>(a) Arrange in Matrix</th>
<th>(b) Transposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 1 5 6 7</td>
</tr>
<tr>
<td>4 1 5 6 7</td>
<td>3 7</td>
</tr>
</tbody>
</table>

(c) Map Relationship

<table>
<thead>
<tr>
<th>new id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>old id</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: 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>

we first arrange the vertices in an 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 4.2(b), we do a simple transpose to the matrix. Next, we relabel all vertices in the row-order. Table 4.2(c) shows the mapping relationship between the new vertex id and original id. In reality, we simply choose $nrow$ 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 $nrow - 1$.

Algorithm 13 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 4.3 shows the iteration number requirement before and after shuffling, with 4 partitions and partition size of 2. Column Original verifies that GraphStone’s execution has fewer iterations than BFS even if we do not do the random shuffling and only do the logical partitioning. And Column Shuffled shows that, after the random shuffling, GraphStone has even better convergence speed.
4.6 Empirical Evaluation

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

4.6.1 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.

**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 them to converge, or 5 times for non-converge ones. For out-of-core graphs, since they require too much time to converge, we set threshold base on vertices activation rates.

**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
Table 4.4: 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 4.5: Summary data showing GraphStone’s speedups over best in class frameworks for In-Memory (a) and Out-of-Core (b) processing.

[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 4.4 shows the properties of those graphs.

4.6.2 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. 4.5 summarizes all our results, with in-memory results compared in Fig. 4.5(a) and out-of-core results compared in Fig. 4.5(b). Fig. 4.5(a) shows speedups over Ligra (the best prior in-memory framework) for a serial, hand-tuned C program, X-Stream, and GraphStone. Fig. 4.5(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. 4.5(b) does not show results for these two approaches.

Fig. 4.5(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. 4.5(b) shows that GraphStone is $11.6 \times$ 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.

4.6.3 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. 4.6(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—i.e., PR and RW where most vertices are accessed for most iterations. GraphStone’s harmonic mean speedup over Ligra is about $4.2 \times$. The maximum speedup is $7.9 \times$, for PageRank. Ligra is especially optimized for sparse traversal algorithms—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. 4.6(b) shows the runtime comparisons on graph rm27. GraphStone achieves a harmonic mean speedup of 4.0× over Ligra. The maximum speedup is 5.7×, achieved on PR. Fig. 4.6(c) shows the runtime comparisons on graph rm24. For this graph, the sparse traversal algorithms finish in about 2 seconds. GraphStone’s harmonic mean speedup is about 1.7× over Ligra. The highest speedup is 2.7× on PR. Because rm24 is a rather
Table 4.5: 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 4.6: 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>

small graph, we find the runtime differences for the a same algorithm may be as high as 0.5 seconds. That’s why we run them for 5 times and get the average. Fig. 4.6(d) compares runtimes for the LiveJournal graph. GraphStone’s harmonic mean speedup is $2.2 \times$ that of Ligra, with a maximum speedup of $6.2 \times$ on PageRank.

For completeness, we include in-memory performance for both X-Stream (in Table 4.5) and a hand-tuned serial C code Table 4.6. 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. 4.6 shows that both Ligra and GraphStone far outperform these approaches on in-memory data.

**Memory Bandwidth Comparisons**  Fig. 4.7(a)-(d) shows memory bandwidth utilization for Ligra and GraphStone for twitter (a), rm27 (b), rm24 (c), and LiveJournal (d). These results show that GraphStone achieves a much better memory bandwidth utilization than Ligra on average. 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.
Figure 4.7: Memory Bandwidth (GB/s) for different graphs.
4.6.4 Out-of-core Experimental Results

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

![Bar charts comparing run times for different algorithms on various graphs processed out-of-core.](image)

**Figure 4.8:** Run times for graphs processed out-of-core.

**Run Time Comparison**  Fig. 4.8 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 the calculation of mean speedup.

Fig. 4.8(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 29.7× over X-Stream. The biggest speedup is achieved on RW which is 60.6×.

Fig. 4.8(b) shows the runtime for the sim26 graph, which is the second largest graph we use. By harmonic mean, GraphStone gets a speedup of 10.1× over X-Stream. The biggest speedup is achieved on RW with 45.0× over X-Stream.

Fig. 4.8(c) shows the runtime for sim13 graph, which is the third largest graph we use. By harmonic mean, GraphStone gets a speedup of 9.6× over X-Stream. The biggest speedup is achieved on RW with 26× 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. 4.9 compares the I/O usage for the 3 large graphs running out-of-core. Fig. 4.9(a) shows the I/O usage for the ccg graph. By harmonic mean, X-Stream makes about 7.1× the I/O access over GraphStone. The biggest I/O reduction is achieved on SSSP with 18.1×. Fig. 4.9(b) shows the I/O usage for the sim26
graph. By harmonic mean, X-Stream uses about 7.4× as much I/O as GraphStone. The biggest I/O difference with X-Stream is achieved on BFS with 15.0×. Fig. 4.9(c) shows the I/O usage for the sim13 graph. By harmonic mean, X-Stream makes about 5.9× as many I/O access as GraphStone. The biggest I/O difference with X-Stream is achieved on BFS with 36.1×. 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 load all edges, or store
messages to disk, which dramatically reduces its required I/Ops compared to X-Stream.

![I/O Bandwidth Comparison](image)

(a) CCG  (b) sim26  (c) sim13

Figure 4.10: I/O bandwidth comparisons.

**I/O Bandwidth Comparison**  Fig. 4.10 compares achieved I/O bandwidth for the 3
large graphs running out-of-core. Fig. 4.9(a) shows the I/O bandwidth for *ccg*. Fig. 4.9(b)
and Fig. 4.9(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,
Table 4.7: Iteration Number for twitter

<table>
<thead>
<tr>
<th>bench</th>
<th>BSP</th>
<th>HYBRID-4</th>
<th>HYBRID-16</th>
<th>HYBRID-64</th>
<th>ASYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>CC</td>
<td>46</td>
<td>35</td>
<td>22</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>SSSP</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>PR</td>
<td>57</td>
<td>56</td>
<td>49</td>
<td>40</td>
<td>35</td>
</tr>
</tbody>
</table>

GraphStone suffers more from imbalance of barriers and does not need to read a lot from disk so its bandwidth is lower than X-Stream. With Fig. 4.10, we conclude that GraphStone’s execution model achieves slightly 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 convergence speed.

4.6.5 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. 4.5).

The results are shown in Table 4.7 for twitter and in Table 4.8 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 that Ligra’s BSP model and is generally competitive with the fully asynchronous code, but achieves much better parallelism.
Table 4.8: Iteration Number for 1j

<table>
<thead>
<tr>
<th>bench</th>
<th>BSP</th>
<th>HYBRID-4</th>
<th>HYBRID-16</th>
<th>HYBRID-64</th>
<th>ASYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>15</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>CC</td>
<td>14</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SSSP</td>
<td>15</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>PR</td>
<td>55</td>
<td>27</td>
<td>23</td>
<td>23</td>
<td>32</td>
</tr>
</tbody>
</table>

4.7 Performance Gain Analysis

4.7.1 In-core Performance Breakdown

Table 4.9: Compare Major features between X-Stream and GraphStone

<table>
<thead>
<tr>
<th>Ligra</th>
<th>GraphStone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relaxed Bulk Sync Processing</td>
<td>Hybrid Model</td>
</tr>
<tr>
<td>– Threads would randomly access a vertex’s latest value or old value</td>
<td>Threads would access vertices’ last iteration’s value in the same partition and all other vertices’ latest value</td>
</tr>
<tr>
<td>Fine-grained Syncs</td>
<td>Barrier</td>
</tr>
<tr>
<td>– Need atomic read-modify-write operations</td>
<td>– No need for fine syncs</td>
</tr>
<tr>
<td>Standard file operations</td>
<td>Mmap</td>
</tr>
</tbody>
</table>

First, we review the main features of 2 systems in Table 4.9. We show the main differences of them are different execution models, different granularity requirements for synchronous operations and different I/O access methods.

We think we have 2 major factors that contribute to the most of performance speedups:

- Hybrid model

- Coarse barriers instead of fine-grained syncs

and a few minor factors for performance speedups:

- Better locality

- Using mmap
• Load imbalance

For Hybrid Model, we get following advantages for its very good converge efficiency:

• Hybrid model requires fewer iterations

• Hybrid model activate fewer vertices in the computation

To prove upper factors, we conduct extra experiments to count iteration numbers and total activated vertices number during each execution, as shown in Table 4.10.

<table>
<thead>
<tr>
<th></th>
<th>BFS</th>
<th>CC</th>
<th>SSSP</th>
<th>PR</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>-44 %</td>
<td>-45 %</td>
<td>-38 %</td>
<td>-36 %</td>
<td>0 % (fixed 5 iterations for both)</td>
</tr>
<tr>
<td>Activated vertices</td>
<td>-16 %</td>
<td>-48 %</td>
<td>+18 %</td>
<td>-36 %</td>
<td>+4 %</td>
</tr>
<tr>
<td>Speedups</td>
<td>44 %</td>
<td>81 %</td>
<td>17 %</td>
<td>62 %</td>
<td>-25 %</td>
</tr>
</tbody>
</table>

From Table 4.10, we could see that iterations and activated vertices both contribute a lot to the execution time. Since we don’t do specific optimization for RW here, imbalance is the major issue for RW here. Also, RW is not a converging algorithm, so we could not measure our hybrid model’s execution efficiency for RW.

Fig. 4.11 shows the possible concurrency issue in Ligra, when 2 threads try to access the same vertex, users have to use some read-modify-write atomic opearations. We handle this case differently in Graphstone, as shown in Fig. 4.12. We use coarse barriers to handle those concurrencies. When thread try to access a vertex A from vertex C in the same partition, we simply use the old value A', so there would no contention at all. And thanks to our shuffle process, it is unlikely that vertex has many neighbors in the same partition, so that we could get converge efficiency close to classic asynchronous model.

For examining the contribute of coarse barriers, we could only simulate the extra instruction cost, because there is no contention in our system. It is too hard to simulate how many contentions actually happen. And contention number vary over multiple runnings.
Figure 4.11: Ligra concurrency hazard

Figure 4.12: Coarse barriers in GraphStone
So the real cost of fine syncs is expected to be higher. We provide a lower bound here with 1.49 X speedup shown in Fig. 4.13.

For localities, GraphStone constrain update to certain number of vertices, normally fit into CPU cache So we get better cache locality. But we cannot directly measure how much speedup we get from here On average, we get 26% fewer MPKI.

For using mmap, we get small speedups. We replace mmap with standard C file operations, and we only get 1.1 X speedup here.

In Fig. 4.13, to study the performance of GraphStone running under in-core mode, we breakdown the performance gain into 3 parts: hybrid model, sync operation saving and mmap. So we add 3 extra evaluations: GraphStone running in BSP mode, BSP + sync operations and BSP + sync + Without Mmap. The breakdown results are shown in Fig. 4.13.

For hybrid model, we could conclude that: i) For algorithms PR and CC having high density of activated vertices, we could save a lot of running time; ii) for algorithms BFS and SSSP having relatively lower density of activated vertices, we get lower performance improvement; iii) For none-converging algorithm RW, we don’t get speedup. And we didn’t tune the partition size, so occasionally GraphStone in hybrid is even slower than in the BSP mode because of the load imbalance by more barriers, under RW. For the locks, we only add most simple sync operations and there is no conflict at all under GraphStone. We could see that those sync operations also contribute a lot to the running time. For mmap operations, we typically get slightly performance gain.

Since we didn’t do special optimization for RW and our methods won’t bring extra performance gain for RW. As shown in Table 4.11, we hereby only summarize the main gains of the 3 factors for the other 4 benchmarks: we get 1.51 × speedup for the hybrid model, 1.49 × gain for synchronous-free, and 1.1 × gain for using mmap. We get about 2.5 × speedup in total. After excluding the RW benchmark, we get about 3.6 × speedups
Table 4.11: Major performance factors breakdown

<table>
<thead>
<tr>
<th>Ligra</th>
<th>GraphStone</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relaxed Bulk Sync Processing</td>
<td>Hybrid Model</td>
<td></td>
</tr>
<tr>
<td>– More iterations</td>
<td>– Fewer Iterations</td>
<td>1.51 ×</td>
</tr>
<tr>
<td>– More activated vertices</td>
<td>– Fewer activated vertices</td>
<td></td>
</tr>
<tr>
<td>Fine-grained Syncs</td>
<td>Barrier</td>
<td></td>
</tr>
<tr>
<td>– High sync overload</td>
<td>– Low sync overload</td>
<td>≥ 1.49×</td>
</tr>
<tr>
<td>– Good load-balance</td>
<td>– Occasionally poorer load-balance</td>
<td></td>
</tr>
<tr>
<td>Standard file operation</td>
<td>Mmap</td>
<td></td>
</tr>
<tr>
<td>– Extra memory load overhead</td>
<td>– No extra copy</td>
<td>1.10 ×</td>
</tr>
</tbody>
</table>

over Ligra. And in the real world that we expect the engine to have more conflicts, and get more performance gain.

4.7.2 Out-of-core Performance Breakdown

To study the performance of GraphStone running out-of-core, we first conclude that the main reason for the performance gain is I/O amount. Both systems get similar I/O throughput for most cases, except the BSP running on sim13 graph and RW. For RW, both systems suffers from bottlenecks from both computation and I/O. To clearly study the benifits of hybrid model, we won’t discuss RW here.

First, we provide some insights here that I/O is a very good indicator for performance. Fig. 4.14 shows the R-sqaure between I/O ratio and performance speedups between those out-of-core experiments excluding BFS on graph sim13, which is the only exception that 2 systems get quite different I/O bandwidths. In this special case, GraphStone seems suffer more from imbalance by barriers.

There are 3 main reasons why X-Stream use much more I/O than GraphStone: i) X-Stream use edge list instead of adjacent list. So X-Stream use $\frac{3}{2} - 3 \times$ I/O compared to GraphStone for a full graph access; ii) X-Stream need more iterations to converge to the same condition as GraphStone; iii) X-Stream has to load all edges for each iteration even if there is only a small number of vertices. But GraphStone allows random access
to small number of vertices.

Fig. 4.8 shows the runtime comparison, Fig. 4.9 shows the amount of I/O used, and Fig. 4.15 shows the iterations used. In Fig. 4.15, we add an special column GraphStone-Amortized to show how much work does GraphStone really do in terms of iterations. But please note that this is a lower bound for I/O GraphStone need. Because GraphStone does random access to block devices, so GraphStone need to load a full block and the kernel would do some extra prefetch. Both of them would incurs some extra cost for loading. But the typically cost is only slightly higher than X-stream for a full access to all edges.

We expect the speedups to be somewhere between column GraphStone and GraphStone-Amortized.

Also, We should be able to predict how much I/O we need by iteration number and total activated vertices/edges. For X-Stream, it will need to load all edges each iteration, and cost for messages is very low, which could be ignore except CC For CC, the messages cost is about the same as loading all edges, so we could simply double the edge cost to get whole cost for CC. For GraphStone, we will load all activated edges. But this is random access, we would actually load more edges than we need. So here we provide an upper bound, which is loading all edges for each iteration, and a lower bound: that GraphStone only loading edges we need. As shown in Table 4.12. In GraphStone column we normally provide an interval of acttual edges we would load, and column X-Stream shows the estimate of edges would be loaded by X-Stream. The last column shows the I/O ratio we measured. And based on this ratio, we find that we have a good estimation about the the I/O used by GraphStone.
Table 4.12: Estimate I/O ratios between GraphStone and X-Stream by iteration number and activate edges

<table>
<thead>
<tr>
<th>Graph-benchmark</th>
<th>GraphStone</th>
<th>X-Stream</th>
<th>Ratio of X-Stream/GraphStone</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim13-bfs</td>
<td>(1.9</td>
<td>E</td>
<td>, 7</td>
</tr>
<tr>
<td>sim13-cc</td>
<td>(18.6</td>
<td>E</td>
<td>, 22</td>
</tr>
<tr>
<td>sim13-dij</td>
<td>(8.2</td>
<td>E</td>
<td>, 18</td>
</tr>
<tr>
<td>sim13-pr</td>
<td>20</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>sim26-bfs</td>
<td>(2</td>
<td>E</td>
<td>, 7</td>
</tr>
<tr>
<td>sim26-cc</td>
<td>(12</td>
<td>E</td>
<td>, 14</td>
</tr>
<tr>
<td>sim26-dij</td>
<td>(8.6</td>
<td>E</td>
<td>, 20</td>
</tr>
<tr>
<td>sim26-pr</td>
<td>19</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>crg-bfs</td>
<td>(0.4</td>
<td>E</td>
<td>, 12</td>
</tr>
<tr>
<td>crg-dij</td>
<td>(1.5</td>
<td>E</td>
<td>, 30</td>
</tr>
</tbody>
</table>
Algorithm 12 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 \textsc{iterate\_once}(cur\_iter)
6: \hspace{1em} while \( cur\_par < max\_par \) do
7: \hspace{2em} for \( partition \in \text{partitions} \) do
8: \hspace{3em} parallel for In-memory Vertex \( v \in \text{partition} \) do
9: \hspace{4em} if \( v \) is \textit{activated} then
10: \hspace{5em} construct corresponding \textbf{Full Vertex} \( v' \) of \( v \) in \textit{patch}
11: \hspace{5em} copy value and scheduling status of \( v \) to \( v' \)
12: \hspace{5em} mark \( v \) as \textit{inactivated}
13: \hspace{3em} end parallel for
14: \hspace{2em} parallel for Full Vertex \( v' \in \text{patch} \) do
15: \hspace{3em} if \( v' \) is \textit{activated} then
16: \hspace{4em} update vertex \( v' \)
17: \hspace{3em} \(\triangleright\) all activation for other vertices made during \textit{update} directly goes to in-memory vertices, so there is no race
18: \hspace{3em} end parallel for
19: \hspace{2em} parallel for Full Vertex \( v' \in \text{patch} \) do
20: \hspace{3em} if \( v' \) is \textit{activated} then
21: \hspace{4em} copy value of \( v' \) back to corresponding In-memory Vertex \( v \) in \textit{partition}
22: \hspace{3em} mark \( v' \) as \textit{inactivated}
23: \hspace{2em} end parallel for
24: end function \textsc{start\_engine}
25: \hspace{1em} cur\_iter \leftarrow 0
26: \hspace{1em} while \( cur\_iter < max\_iter \) and \( need\_more\_iteration \) do
27: \hspace{2em} need\_more\_iteration \leftarrow false
28: \hspace{2em} \textsc{iterate\_once}(cur\_iter)
29: \hspace{1em} cur\_iter \leftarrow cur\_iter + 1

Algorithm 13 Mapping Function from Shuffled ID to Actual ID

1: Global Variables:
2: \( size\_t \ nrow; \)
3: \( size\_t \ ncol; \)
4: function \textsc{Map2NewId}(oldid)
5: \hspace{1em} return \( \text{oldid} \% \ nrow \times \ ncol + \text{oldid}/\text{nrow} \)
6: function \textsc{Map2NewId}(newid)
7: \hspace{1em} return \( \text{newid} \% \ ncol \times \ nrow + \text{newid}/\text{ncol} \)
Figure 4.13: Run times breakdown for graphs processed in memory.
Figure 4.14: R-square between speedups and I/O ratios.
Figure 4.15: Iteration number comparisons.
CHAPTER 5
FUTURE WORK AND CONCLUSION

5.1 Conclusion

This thesis has presented 2 graph systems for processing out-of-core graphs on single machines. GraphZ has a new programming model and two methods of improving large-scale graph analytics on small-scale systems. The first method is a novel storage format for graphs, which we call degree-ordered storage. Degree-ordered storage reduces the memory footprint of the graph allowing more vertices to fit into memory at once. The second method is a new programming model based on ordered dynamic messages, which allows messages to be immediately intercepted and applied to the destination. This messaging model reduces the storage required for message data, and increases parallelism. Our experimental results confirm the hypothesis that these optimizations reduce IO pressure and increase performance. Compared to other state-of-the-art solutions, GraphZ can significantly reduce runtime for large graphs which must be processed out-of-core. We release GraphZ as an open-source project so that others can build on these results or compare to them.

GraphStone is unified graph analytics framework that achieves best-in-class performance for both in-memory and out-of-core processing. GraphStone overcomes the challenges of achieving performance across these regimes by combining a hybrid synchronous execution model with on-the-fly vertex construction and message generation. These innovations allow GraphStone programs to reduce the number of iterations required for convergence, while achieving high memory bandwidth and IO utilization. GraphStone’s authors are thankful to the Ligra, GraphChi, and X-Stream developers for releasing their code as open source and allowing us to easily compare with their approach. In return, we release GraphStone as open source, with the hope that others will improve on it in the
future.

5.2 Future Work

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. There are a few work already working on approximate computation.

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 approximately 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. But there is no general approximate graph computation solution that provides a general framework for major graph analytic algorithms and provide correctness studies. People could work more on this and give some general solutions.
APPENDIX 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. 2.1(a) is 0–6, so its **path iteration length** is also 6. But for ASYN on Fig. 2.1(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 2.3(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 2.3(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 \( \{v_0, v_1, v_2, ..., v_{l-1}\} \) 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 \( \{v_0, v_1, v_2, ..., v_{l-1}\} \). Given an
iteration method \( I \in \{BSP, ASYN, HYBRID\} \), we define a function \( PIL_I(l) \) to get the
expected PIL of a random path of length \( l \), and function \( PIL_I(l, v) \) 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:

\[
PIL_{BSP}(l) = l
\]
For ASYN, we recursively compute the PIL:

\[
PIL_{ASYN}(l, v) = \sum_{i=0}^{v-1} PIL_{ASYN}(l-1, i) + \sum_{i=v+1}^{\lfloor V \rfloor-1} (PIL_{ASYN}(l-1, i) + 1)
\]

\[
PIL_{ASYN}(l) = \frac{\sum_{v=0}^{\lfloor V \rfloor-1} PIL_{ASYN}(l, v)}{|V|}
\]

To derive \( PIL_{ASYN}(l) \), we use \( PIL_{ASYN}(l, v) \) 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 and \( |V| - 1 \) is the same, so we can just sum them up and divide by \( |V| \). We also use \( PIL_{ASYN}(l-1, v) \) to recursively get \( PIL_{ASYN}(l, v) \). 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 \( PIL_{HYBRID}(l, p) \) 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:

\[
PIL_{HYBRID}(l, p) = \sum_{i=0}^{p-1} PIL_{HYBRID}(l-1, i) + \sum_{i=p}^{np-1} (PIL_{HYBRID}(l-1, i) + 1)
\]

\[
PIL_{HYBRID}(l) = \frac{\sum_{p=0}^{np-1} PIL_{HYBRID}(l, p)}{np}
\]
Similar to $PIL_{ASYN}(l)$, we use $PIL_{HYBRID}(l, p)$ 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 $PIL_{HYBRID}(l-1, p)$ to recursively get $PIL_{HYBRID}(l, p)$. 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 A.1: 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>1.0</td>
<td>1.0</td>
<td>2.1</td>
<td>1.3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2.0</td>
<td>2.0</td>
<td>4.3</td>
<td>5.1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4.0</td>
<td>8.1</td>
<td>8.5</td>
<td>10.3</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>8.0</td>
<td>16.3</td>
<td>17.0</td>
<td>20.6</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>16.0</td>
<td>32.5</td>
<td>34.1</td>
<td>41.1</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>32.0</td>
<td>65.0</td>
<td>68.1</td>
<td>82.3</td>
</tr>
</tbody>
</table>

Table A.1 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 A.1 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.
BIBLIOGRAPHY


[52] K. Vora, G. Xu, and R. Gupta. “Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing”. In: 2016 USENIX Annual Techni-


