GraphZ: Improving the Performance of Large-Scale Graph Analytics on Small-Scale Machines

Zhixuan Zhou Henry Hoffmann
University of Chicago, Dept. of Computer Science

***Abstract*—Recent programming frameworks enable small computing systems to achieve high performance on large-scale graph analytics by supporting *out-of-core* graph analytics (i.e., processing graphs that exceed main memory capacity). These frameworks make out-of-core programming easy by automating the tedious process of scheduling data transfer between memory and disk. This paper presents 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.**

I. Introduction

Graph analytics are key computing workloads; however, many graph data sets are too large for a single computer's memory. For example, Facebook reports internal processing of trillion-edge graphs [9]. Two classes of specialized programming systems address large-scale graph analytics. The first *distributes* graph processing, using multiple machines' memory to analyze a single graph [2, 3, 14, 17, 27, 28, 30, 35, 43]. The second processes graphs *out-of-core*, automatically orchestrating data movement between memory and backing store [16, 19, 33, 53]. Remarkably, these out-of-core systems achieve performance comparable to, or better than, distributed systems [19]. For both approaches, however, off-chip IO is the primary performance limiter.

Whether distributed or out-of-core, most graph programming frameworks expose a *gather-apply-scatter* model [27]. Programmers specify an `update` function that is called on each vertex and *messages* transfer data between vertices. When a `update` is called, it *gathers* all inbound messages, *applies* the `update` function to those messages and the local vertex data, and then *scatters* new messages to adjacent vertices. For example, in PageRank [20], each update gathers the votes of adjacent vertices, computes a new rank for the current vertex, and scatters this new rank to neighbors. The key to ease-of-use is that the runtime tracks all vertex and edge locations; automatically moving data either across a distributed system or between memory and backing store, relieving software engineers from this tedious task.

To automate data movement, the runtime maintains a *vertex index* that maps each vertex to the location of its adjacency list on disk. Using typical storage formats (e.g., compressed sparse rows [38] or prefix sum [13]), these indices need space proportional to the number of vertices. While vertex storage is small compared to edge storage, the index is still crucial because an index larger than memory requires two disk accesses per vertex access: one to load the the index and one to load the vertex data itself. To avoid these IOs and save index storage, we introduce *degree ordered storage*: a new format that takes space proportional to the number of unique degrees. Degree ordered storage is compact—all vertices of the same degree are indexed with a single number and a pointer to the location of the vertex with the next degree. As natural graphs tend to have a small number of unique degrees [14], this format saves tremendous space for the vertex index, reducing IO and processing time.

Degree ordered storage is also a general way for improving data locality of natural graphs. Previous approaches to improving locality rely on graphs' internal properties like dictionary order, label adjacency and geometry adjacency [4, 7, 25, 39, 49, 54]. These locality properties typically apply to a just a narrow range of graphs and are not applicable to graphs of other types. Natural graphs, however, follow power-law distributions [14], and degree ordered storage naturally arranges high-degree vertices together in memory. Thus messages between these vertices achieve high locality, reducing IO operations and decreasing processing time.

Messages are the key construct for implementing the gather-apply-scatter model. Messages, however, must be stored until a subsequent update gathers them, consuming both memory and IO bandwidth. Some programming systems, like Pregel, allow *hints* suggesting how messages could be combined or reduced to save IO [28]. Unfortunately, there are no guarantees the hints will be applied and there is no control over their ordering if applied. This approach, thus, has the twin drawbacks of reducing programmer control and forcing the runtime to allocate a large amount of intermediate storage.

To provide message ordering while reducing space requirements, we propose *ordered dynamic messages*. Inspired by Active Messages [12], ordered dynamic messages specify both data transfer and computation to perform at the destination; i.e., they combine both the gather and apply steps from the gather-apply-scatter model. By computing on the data as soon as it is available the runtime removes much intermediate message storage, allowing more of the graph into memory at once, reducing IO and run time. While ordered dynamic messages are motivated by the observation that most graph analytics computations use commutative and associate operators during the apply phase, we note that is not a requirement. In fact, we show that a system build with ordered dynamic messages is at least as expressive as existing frameworks.

We combine degree-ordered storage and ordered dynamic messages into a new graph computing framework called GraphZ, which we implement in C++ for Linux/x86. We test GraphZ on six common graph analytics benchmarks using different graph sizes and both an SSD and HDD. We compare GraphZ to two state-of-the-art out-of-core approaches: GraphChi [19] and X-Stream [33] to show:

- Converting graphs to degree ordered storage is faster than converting to other common formats.
- When graphs fit in memory, there is no best system.
- For a graph that just exceeds memory, GraphZ provides mean speedups 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 speedups are 1.8-4.8× faster than GraphChi and 1.8-3× faster than X-Stream. Individual speedups are as high as 8× and 13× compared to GraphChi and X-Stream.
- The reduction in IO and runtime produces dramatic reduction in energy consumption as well. By harmonic mean GraphZ consumes only 45% the energy of GraphChi and just 40% the energy of X-Stream.
- For a graph that exceeds SSD capacity, GraphChi fails, but GraphZ's achieves 1.8× speedup over X-Stream.

GraphZ contributes the **degree-ordered storage** format that reduces memory requirements and disk accesses; and **ordered dynamic messages** that reduce IO operations. We release GraphZ and benchmarks as open source so that others may recreate or extend these results.[1]

II. Background and Related Work

This section covers two areas of related work: 1) existing graph programming systems and 2) a recent study questioning the need for any such programming system [29].

A. Existing Graph Programming Systems

Many graph systems distribute data across machines. Giraph processes graphs on Hadoop [2]. Pregel is a customized, distributed graph engine [28]. Other distributed approaches include GraphLab [27], PowerGraph [14], the Parallel Boost Graph Library [15], and Hama [3]. Subsequent work improves performance of these custom systems, especially through better distributed load balancing [17, 35], combining synchronous and asynchronous execution models [30], and increasing graph processing fault tolerance [43]. Chaos is one of the few distributed graph processing systems that combines distribution with use of secondary storage [34].

[1]https://github.com/danden89/GraphZ

Instead of distributing, some graph systems work out-of-core. GraphChi supports single-node out-of-core graph analytics and *achieves performance equal to or better than distributed systems* [19]. GraphChi's runtime automatically manages data movement between disk and memory. GraphChi uses an *asynchronous* execution model [6], meaning that it always performs updates using the most recent vertex data. Compared to a *bulk-synchronous model*, where all vertices are updated at once, the asynchronous model generally reduces iterations and run time [19]. The PowerLyra system, however, achieves even better performance by mixing both asynchronous and bulk-synchronous processing [8].

Other single-machine systems fulfill various niches. TurboGraph customizes for high-end hardware [16]. FlashGraph works on single-node system backed by an array of high-end SSDs [52]. X-Stream presents an edge-centric programming model based on the observation that edges outnumber vertices, so edge accesses should be as fast as possible [33]. An extension of the edge-based model focuses on reducing disk IOs by carefully loading only those edges that will be used – a method which reduces IO but not runtime compared to X-Stream [41]. GridGraph extends the edge-centric model using a two-level partitioning scheme [53]. While the edge-centric scheme improves performance, it results in more complicated programs (see Table IX) and requires the bulk-synchronous execution model.

Other projects focus on IO reduction. PathGraph adopts a path-centric approach, so programmers focus not on vertices, but on vertex traversal order [51]. GraphQ focuses on sub-graph queries, so programmers only manipulate subsets of graphs [42]. GraphQ saves time when queries can be answered using only a few sub-graphs, but not all queries meet this criteria. A recent modeling effort estimates graph traversal locality, which complements these approaches by finding the most efficient paths ahead of time [50]. MMap automatically shuffles data between memory and storage with far less code required than most graph frameworks [24].

Several approaches manipulate vertex ordering for performance. Ligra allows users to specify ordering to combine both top-down and bottom-up graph traversals, but Ligra is not for out-of-core processing [36, 37]. The Galois framework provides iterators for irregular computations. It is more general than graph-specific approaches and this generality improves performance in some cases [31]. Another effort shows that the order of iterative updates can make up to a 5× difference in runtime for symmetric computations, like triangle counting [1]. While all these approaches use ordering to improve performance, GraphZ's degree ordered storage is unique in that it is the only approach to use ordering to compact the vertex index required for out-of-core processing. Future work may combine the two approaches to both produce more compact approaches that lead to faster convergence.

This paper improves out-of-core graph analytics performance. GraphZ maintains the widely adopted vertex-centric model, while adding support for degree-ordered storage and ordered dynamic messages. Like GraphChi and X-Stream, we

TABLE I: Lines of Code to Implement Page Rank.

Graph Size	C	GraphChi	GraphZ
in memory	300	23	24
out-of-core	500	23	24

TABLE II: Time to Execute Page Rank.

Graph Size	C	GraphChi	GraphZ
in memory	6s	22s	16s
out-of-core	3848s	2958.2	2024.3s

release GraphZ as open-source.

B. Graph Programming Systems' Benefits

McSherry et al. question the need for any graph programming system [29]. 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. VI. In brief, we use the LiveJournal graph [47] to test in-memory performance and YahooWeb graph [45] to test out-of-core performance.

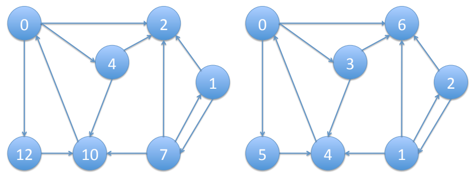
Table I shows the LOC required to implement PageRank, while Table II shows the runtime. We conclude:

- 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 [29].
- 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 [9], we believe the need for out-of-core frameworks will only become greater.

III. 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.



(a) Original Graph (b) Relabeled Graph

Fig. 1: Example of Relabeling and tight ID slots

TABLE III: Example Graph

source	dest	degree
0	2, 4, 12	3
1	2, 7	2
2		0
4	2, 10	2
7	1, 2, 10	3
10	0	1
12	10	1

A. 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 [38], 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 [14], so this format requires a (typically) much smaller number of indices than vertices, greatly reducing index size compared to CSR.

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

B. Example

We illustrate degree-ordered storage using the graph in Figure 1a. Table III 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 [10, 21–23, 48].

TABLE IV: Relabeling of Example Graph

src	dest	degree	new src id	new dst id
0	2, 4, 12	3	0	3, 5, 6
7	1, 2, 10	3	1	2, 4, 6
1	2, 7	2	2	1, 6
4	2, 10	2	3	4, 6
10	0	1	4	0
12	10	1	5	4
2		0	6	

TABLE V: Edge List Stored on External Disk

offset	0	1	2	3	4	5	6	7	8	9	10	11
edges	3	5	6	2	4	6	1	6	4	6	0	4

We first sort the vertices by descending out-degree, with ties broken randomly. Table IV 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 1b shows the relabeled graph. Having relabeled the graph, we store: the map between the old and new IDs (columns 1 & 4 from Table IV), the ordering on the adjacency lists (Table V), 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 VI 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 VII 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 $ids\_table[d]$. Then look in the id_offset_table to get the offset of vertex $ids\_table[d]$. Finally we compute the offset of vertex x, by the formula:

$$offset = id\_offset\_table[d] + (x - ids\_table[d]) \times d \qquad (1)$$

For example, to find the offset of vertex 3, we do a binary search on 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 $id\_offset\_table[2]$ 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.

C. Implementation

The conversion to DOS requires only sequential access to external storage. We convert the graph to a new list, $EDGES$,

TABLE VI: ids_table

degree	3	2	1	0
id	0	2	4	6

TABLE VII: id_offset_table

degree	3	2	1	0
offset	0	6	10	12

of $triad\langle src, dest, deg\rangle$, where deg is the degree of src. Then we use external k-way merge sort [11, 18, 40] to sort it using deg as 1st key and src as 2nd key. Thus, we can relabel all those srcs, with sequential access. At the same time we record the mapping from $newid$ to $oldid$. After this, we get a list of $pair\langle newid, oldid\rangle$, then we do another sort by key $oldid$, and we record this sequential mapping from $oldid$ to $newid$ on external storage.

Next, we sort $EDGES$ by $dest$. Thus, with the mapping from $oldid$ to $newid$, we sequentially relabel $dest$s in $EDGES$. 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 $EDGES$ by src again, and generate all those support files for each partitions. deg in $EDGES$ 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; *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. VI-B).

D. 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: $|UD| \leq 2\sqrt{|E|}$

Prove 1. We divide UD into subsets UD_1 and UD_2.
Let $UD_1 = \{d \in UD | d < \sqrt{|E|}\}$. Then, $|UD_1| \leq \sqrt{|E|}$.
Let $UD_2 = \{d \in UD | d \geq \sqrt{|E|}\}$.
Let $V_2 = \{v \in V | degree(v) \geq \sqrt{|E|}\}$.
Function $degree(v)$ returns the degree of vertex v.
Assume $|UD_2| > \sqrt{|E|}$, then

$$\begin{aligned}|E| &= \sum_{v \in V}^{v} degree(v) \geq \sum_{v \in V_2}^{v} degree(v) \geq \sum_{v \in V_2}^{v} \sqrt{|E|} \\ &= |V_2| \times \sqrt{|E|} \geq |UD_2| \times \sqrt{|E|} \\ &> \sqrt{|E|} \times \sqrt{|E|} = |E|\end{aligned}$$

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

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

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

TABLE VIII: SNAP graph properties.

	Graphs				
	as-skitter [21]	cit-Patents [21]	com-orkut [48]	higgs-twitter [10]	wiki-Talk [22, 23]
Vertices	1.7M	3.8M	3M	457K	2.4M
Edges	22M	17M	234M	15M	5M
Unique degrees	2.0K	0.7K	5.4K	1.7K	1.7K

This analysis backs up the notion that the number of unique degrees in a graph must be small. Additionally, Table VIII 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.

E. Analysis of Edge Density Distribution

In addition to reducing the storage requirement for 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. To make the graph more clear, we only show the in-partition messages in the first partition. In-partition messages in other partitions are also high. 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 the highest degrees, compared to other partitions.

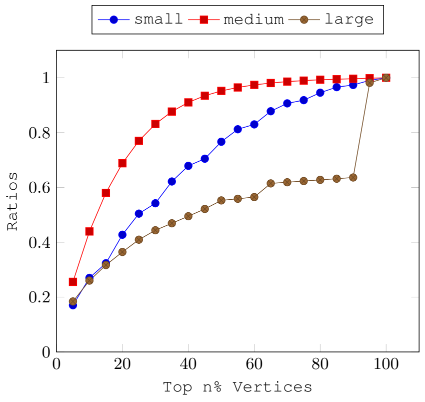


Fig. 2: CDF of in-partition messages as a function of partition size.

Fig. 2 shows the ratio of edges within the top $n\%$ of vertices for natural graphs used in our evaluation (see Sec. VI-A). 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. The `large` graph looks a little different from the other 2, because some low degree vertices were removed from the real world. If it is complete, we could expect it to show the same curve as other 2.

IV. 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 [19] and X-stream [33], existing open-source, out-of-core graph programming systems.

A. 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 1, 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 $float$s – 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.

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 1 Key Data Structures

```
struct VertexDataType {
    int $vval1$;
    float $vval2$;
    double $vval3$;
    .....                          ▷ it could be $int$, $float$, $struct$, $array$....
                     ▷ Value initiation could be done under $constructor$
}
struct MessageDataType {
    int $msg\_val1$;
    float $msg\_val2$;
    double $msg\_val3$;
    .....                          ▷ it could also be $int$, $float$, $struct$, $array$....
}
```

Algorithm 2 User-defined methods in GraphZ

```
function UPDATE($vertex$)
    $vertex \leftarrow f_1(vertex)$                          ▷ not a must
    for $vadj$ in $vertex$'s adjacent $vertices$ do
        if some condition then
            compute a message $msg$
            $send$ the message $msg$ to $vadjv$
function APPLY_MESSAGE($vertex$, $msg$)
    $vertex \leftarrow f_2(vertex, msg)$
    return vertex
▷ $f_2$ is often a very simple function, like $min(vertex, msg)$, $vertex$ + $msg$, $vertex$.append($msg$)
```

B. Execution Model

GraphZ applies the `update()` methods using the asynchronous execution model shown in Figure 3.

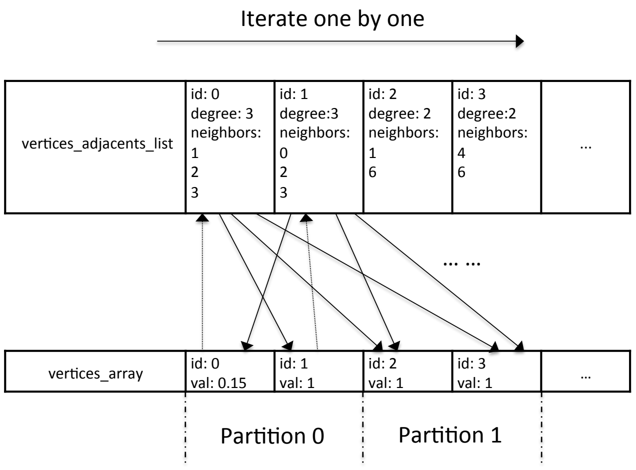


Fig. 3: Graph Processing in the View of Developers

GraphZ's runtime manages two lists: `vertices_array` and `vertices_adjacents_list`. `vertices_array` stores all vertices' values and `vertices_adjacents_list` stores every vertex's adjacency list in degree-sorted order (see Sec. III).

GraphZ's runtime iterates through `vertices_array`, calling `update()` on every vertex. Fig. 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-

Algorithm 3 PageRank Data Structures

```
struct VertexDataType {
    float $vval = 1$;
    float $votes = 0$;
}
struct MessageDataType {
    float $msg$;
}
```

Algorithm 4 PageRank

```
function UPDATE($vertex$)
    $ndeg \leftarrow number\ of\ vertex's\ adjacent\ vertices$
    $vertex.vval \leftarrow 0.15 + 0.85 * vertex.votes$
    $vertex.votes \leftarrow 0$
    if $ndeg == 0$ then return
    if $cur\_iter == 0$ then
        $msg \leftarrow \frac{1}{ndeg}$
    else
        $msg \leftarrow \frac{vertex.vval}{ndeg}$
    for $vadj$ in $vertex$'s adjacent $vertices$ do
        $send$ message $vertex.vval$ to $vadj$
function $apply\_message$ ($vertex$, $msg$)
    $vertex.votes \leftarrow vertex.votes + msg$
return vertex
```

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.

C. 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 v_1 and v_2, if $o(v_1) < o(v_2)$, then the `update()` method will be called on v_1 first in every iteration and messages sent during the update execution of v_1 are also applied before updating v_2. Besides ease-of-use, maintaining *consistency* also has a performance advantage, which can greatly accelerate the convergence speed and reduce disk accesses [19]. To be clear, the user does not get to specify the order, which is determined by GraphZ's degree-ordered storage format (see Sec. III). 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.

D. PageRank Example

We illustrate PageRank in GraphZ. On initialization, we assign all vertices rank 1 and received votes to 0. In Equation 2[32], vertices B, C, D and etc. are

$$PR(A) = 1 - d + d(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + ...) \quad (2)$$

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 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()`.

E. 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

```
struct Edge {
    vertex_id neighbor;
    EdgeDataType edge_val;
}
struct VertexDataType {
    List <Edge> edges;
    RealVertexDataType vertex_val;
}
struct MessageDataType {
    Edge edge;
}
```

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.

F. 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

Algorithm 6 Covert GraphChi Programs to GraphZ

```
function UPDATE($vertex$)
    for $edge$ in $vertex$'s $edges$ do
        read edge data from $edge$
        if some condition then
            update $vertex\_val$
                ▷ this self update could also be done outside this $for$ loop
    for $vadj$ in $vertex$'s adjacent $vertices$ do
        Construct a $msg$
        $msg.edge.neighbor \leftarrow vertex's\ id$
        if some condition then
            compute a message $real\_msg$
            $msg.edge.edge\_val \leftarrow real\_msg$
        $send$ the message $msg$ to $vadjv$
    $vertex.edges \leftarrow \emptyset$
function $apply\_message$ ($vertex$, $msg$)
    $vertex.edges$.append($msg.edge$) return vertex
```

TABLE IX: LOC Comparison of Graph Engines.

Benchmark	GraphChi	X-Stream	GraphZ
BFS	34	99	25
CC	32	64	13
PR	23	60	24
BP	30	254	50
RW	30	65	30
SSSP	32	59	30

the LOC (lines of code) required to write our test algorithms in GraphZ, GraphChi, and X-Stream.

Table IX shows the LOC needed for the six algorithms used in our experimental evaluation (see Sec. VI-A). 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.

V. IMPLEMENTATION

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

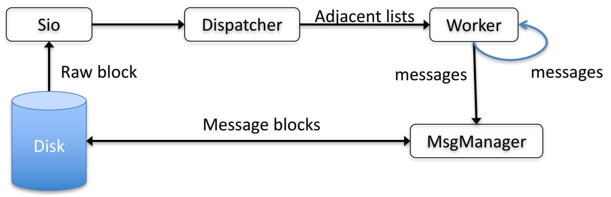


Fig. 4: GraphZ Implementation Overview

Fig. 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 `apply_message()` on any destination vertices within the partition, enforcing the message ordering. Next, Sio reads the graph storage file into memory and reads *edge blocks* that are passed to the Dispatcher, which translates them into adjacency lists. The Worker calls `update()` on each vertex in the partition. The Worker also checks each message's destination and calls `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.

A. 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 adjacency 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 $len/(r-l)/sizeof(Vertex\_ID)$ edges. The Dispatcher allocates memory for each vertex's adjacency list.

B. 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.)

```
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. IV-B. 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 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.

C. MsgManager & Dynamic Messages

The MsgManager has two jobs. First, during updates to the current partition, the Msg Manager 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.)

```
1: procedure MSGMANAGER
2:     while unprocessed messages exist do
3:         read a message $msg$
4:         determine $msg$'s target vertex's partition number $partition\_id$
5:         write $msg$ to partition's buffer with $partition\_id$
```

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.

VI. 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

TABLE X: Graph Properties.
(M = Million, B = Billion, GB = Gigabytes.)

Graph	**LiveJournal**	**Friendster**	**YahooWeb**	**Sim**
Size	small	medium	large	xlarge
Vertices	4M	124.8M	1.4B	3.9B
Edges	69M	3.6B	6.6B	26.2B
Size	560MB	27.8GB	60.0GB	224.4GB
Unique degrees	1.3K	3.1K	2.0K	47.4K

TABLE XI: Vertex index size executing PageRank.

	Graphs			
	small	medium	large	xlarge
GraphChi	30.8MB	952.4MB	10.5GB	28.8GB
GraphZ	43KB	49KB	32KB	758KB

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.

A. 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. We measure power and energy with a WattsUp power meter that caputers full system power and energy consumption at 1s intervals.

Graph Algorithms: We use 6 benchmarks: Connected Components (CC), Breadth-first search (BFS), PageRank (PR) [20, 32], Single-Source Shortest Paths (SSSP), Belief Propagation (BP) [5, 44], and Randomwalk (RW) [26]. GraphChi comes with CC, PageRank, and Randomwalk, while X-Stream lacks Randomwalk. We implement the missing algorithms. Some combinations of benchmarks and storage cause errors for GraphChi or X-Stream. If we are unable to obtain a particular result, that entry in the charts is blank and it is not included when we compute aggregate statistics.

Inputs: Table X shows the basic properties, including number of edges, vertices, and storage requirement, of the 3 natural graphs and 1 synthetic graph (generated according to [46]) used in this study: LiveJournal [47] (small), Friendster [47] (medium), YahooWeb [45] (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 XII: Preprocessing time (s).

	Graphs						
	small		medium		large		xlarge
	HDD	SSD	HDD	SSD	HDD	SSD	HDD
GraphChi	18	17	3193	1422	6413	2613	42240
GraphZ	17	17	3040	1102	5969	2299	29561
X-Stream	265	258	13701	13402	25046	24809	99124

B. Preprocessing And Vertex Index Size

To demonstrate degree ordered storage's benefits, Table XI 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 XII 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.

C. Extra Large Graph Performance

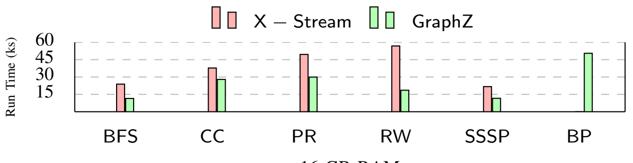


Fig. 5: Run time comparison on the xlarge graph.

D. Large Graph Performance

The results for the extra large graph (Sim) on HDD are shown in Fig. 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 XI). 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.

The results for the large graph (YahooWeb) are shown in Fig. **??**. 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

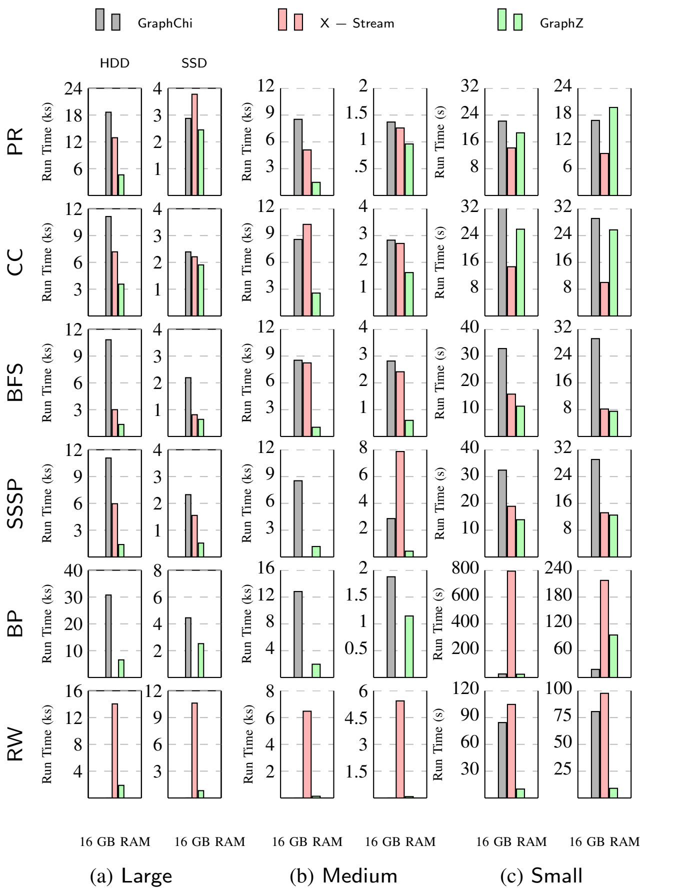


Fig. 6: Run times for the medium graph.

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, however.

E. Medium Graph Performance

Fig. 6 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.

The results for the small graph are shown in Fig. **??**. 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.

TABLE XIII: Relative Energy Consumptions

	large	medium	small
GraphZ/GraphChi	0.45	0.29	0.48
GraphZ/X-Stream	0.4	0.32	0.74

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. 6(b), however, shows the results of in-memory graph processing with GraphZ are competitive or even sometimes much better than existing approaches.

F. Performance Breakdown

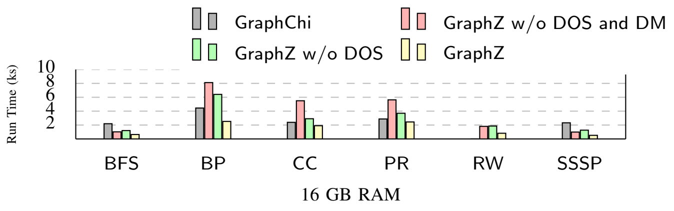


Fig. 7: Performance breakdown for the large graph.

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

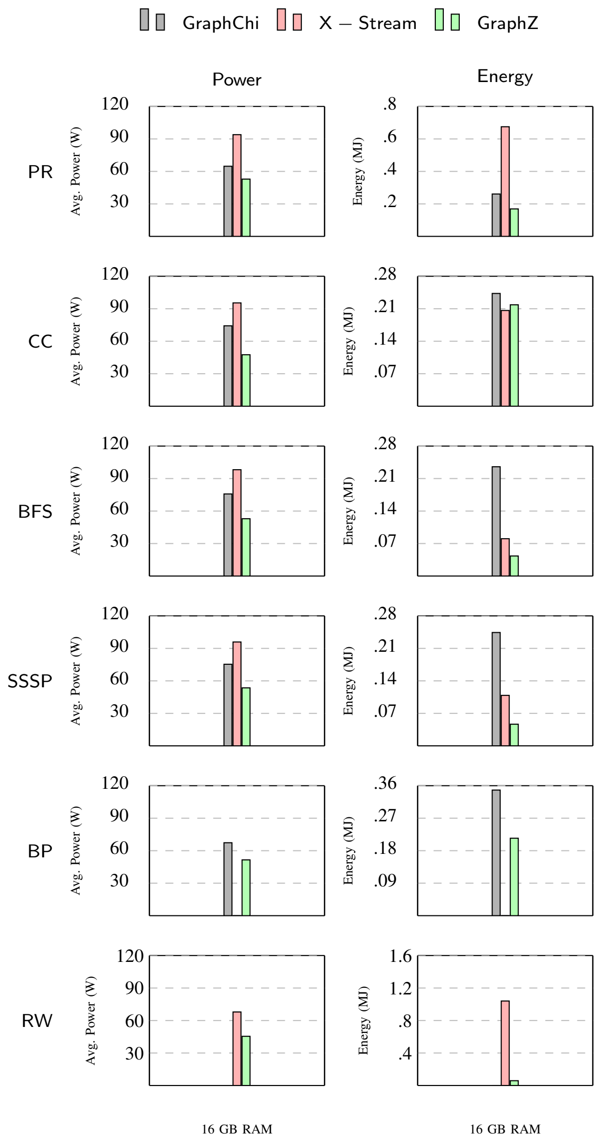


Fig. 8: Power and energy for large graph.

G. Energy Reduction

Fig. 8 shows power in Watts (on the left) and energy in Megajoules (on the right) for the large graph using the SSD and 16GB of RAM. Power and energy are measured with a WattsUp device and include the total system power consumption (including memory, processor, disks, fans, etc.).

The left set of charts shows that GraphZ has generally lower power consumption than the other frameworks. This lower power combined with the reduced runtime (seen in Fig. **??**) results in large energy savings for GraphZ compared to the other frameworks. Using harmonic mean, GraphZ consumes only 45% the energy of GraphChi and 40% the energy of X-Stream. This is a tremendous energy reduction for the same computation.

Due to space limitations, we cannot include the bar charts for all graphs and benchmarks. Table XIII shows summary data with the relative energy consumption of GraphZ compared to both GraphChi and X-Stream. For the medium graph, GraphZ achieves over $3\times$ savings compared both of the other approaches. Additionally, while the small graph showed no distinct advantages in terms of performance, GraphZ does provide a clear win in terms of energy reduction, consuming less then 50% of the energy of GraphChi and just 74% of the energy of X-Stream. Thus, GraphZ has even greater energy savings than might be predicted from the speedup alone.

TABLE XIV: Iterations for Convergence

		SSSP	**CC**	**BFS**
Small	GraphChi	7	7	7
	X-Stream	47	15	15
	GraphZ	8	8	7
Medium	GraphChi	10	10	10
	X-Stream	59	24	24
	GraphZ	12	11	10

H. 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 XIV 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.

I. 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. 9 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

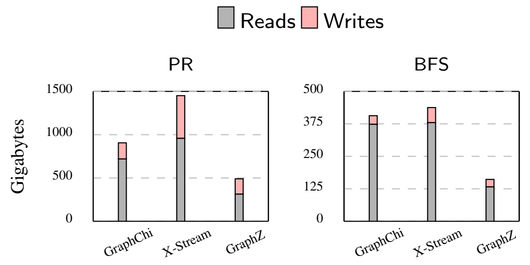


Fig. 9: Total IO volume for large graph

X-Stream. This data confirms that GraphZ's model provides tremendous IO reduction compared to other state-of-the-art approaches.

VII. Conclusions

This paper has presented 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.

References

[1] C. R. Aberger et al. "EmptyHeaded: Boolean Algebra Based Graph Processing". In: *CoRR* abs/1503.02368 (2015).

[2] *Apache Giraph*. URL: http://giraph.apache.org.

[3] *Apache Hama*. URL: http://hama.apache.org.

[4] S. Beamer et al. "Locality exists in graph processing: Workload characterization on an ivy bridge server". In: *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 56–65.

[5] *Belief Propagation*. URL: http://en.wikipedia.org/wiki/Belief_propagation.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[7] P. Boldi et al. "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks". In: *Proceedings of the 20th international conference on World wide web*. ACM. 2011, pp. 587–596.

[8] R. Chen et al. "PowerLyra: differentiated graph computation and partitioning on skewed graphs". In: *EuroSys*. 2015.

[9] A. Ching et al. "One Trillion Edges: Graph Processing at Facebook-Scale". In: *PVLDB* 8.12 (2015), pp. 1804–1815. URL: http://www.vldb.org/pvldb/vol8/p1804-ching.pdf.

[10] M. De Domenico et al. "The anatomy of a scientific rumor". In: *Scientific reports* 3 (2013).

[11] R. Dementiev and P. Sanders. "Asynchronous Parallel Disk Sorting". In: *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '03. San Diego, California, USA: ACM, 2003, pp. 138–148. ISBN: 1-58113-661-7.

[12] T. von Eicken et al. "Active Messages: A Mechanism for Integrated Communication and Computation". In: *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*. Gold Coast, Australia, 1992, pp. 430–440.

[13] T. Goldberg and U. Zwick. "Optimal deterministic approximate parallel prefix sums and their applications". In: *ISTCS*. 1995, pp. 220–228.

[14] J. E. Gonzalez et al. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30.

[15] D. Gregor and A. Lumsdaine. "The parallel bgl: A generic library for distributed graph computations". In: *POOSC*. 2005.

[16] W.-S. Han et al. "TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC". In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, 2013, pp. 77–85.

[17] Z. Khayyat et al. "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 169–182.

[18] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[19] A. Kyrola et al. "GraphChi: Large-Scale Graph Computation on Just a PC". In: *OSDI*. Hollywood, CA: USENIX, 2012, pp. 31–46.

[20] Lawrence et al. *The PageRank Citation Ranking: Bringing Order to the Web*. Tech. rep. 1999-66. Stanford InfoLab, 1999.

[21] J. Leskovec et al. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations". In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. Chicago, Illinois, USA: ACM, 2005, pp. 177–187. ISBN: 1-59593-135-X.

[22] J. Leskovec et al. "Predicting positive and negative links in online social networks". In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 641–650.

[23] J. Leskovec et al. "Signed networks in social media". In: *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM. 2010, pp. 1361–1370.

[24] Z. Lin et al. "Mmap: Fast billion-scale graph computation on a pc via memory mapping". In: *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE. 2014, pp. 159–164.

[25] N. Linial. "Locality in distributed graph algorithms". In: *SIAM Journal on Computing* 21.1 (1992), pp. 193–201.

[26] L. Lovász. "Random walks on graphs: A survey". In: *Combinatorics, Paul erdos is eighty* 2.1 (1993), pp. 1–46.

[27] Y. Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727.

[28] G. Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146.

[29] F. McSherry et al. "Scalability! But at what COST?" In: *HotOS*. 2015.

[30] J. Nelson et al. "Crunching Large Graphs with Commodity Processors". In: *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*. HotPar'11. Berkeley, CA: USENIX Association, 2011, pp. 10–10.

[31] D. Nguyen et al. "A lightweight infrastructure for graph analytics". In: *SOSP*. 2013.

[32] *PageRank*. URL: http://en.wikipedia.org/wiki/PageRank.

[33] A. Roy et al. "X-Stream: edge-centric graph processing using streaming partitions". In: *SOSP*. 2013, pp. 472–488.

[34] A. Roy et al. "Chaos: Scale-out Graph Processing from Secondary Storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 410–424.

[35] Z. Shang and J. X. Yu. "Catch the wind: Graph workload balancing on cloud". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pp. 553–564.

[36] J. Shun and G. E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory". In: *PPoPP*. 2013.

[37] J. Shun et al. "Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+". In: *DCC*. 2015.

[38] *Sparse Matrix*. URL: http://en.wikipedia.org/wiki/Sparse_matrix.

[39] J. Ugander et al. "The anatomy of the facebook social graph". In: *arXiv preprint arXiv:1111.4503* (2011).

[40] J. S. Vitter. "External Memory Algorithms and Data Structures: Dealing with Massive Data". In: *ACM Comput. Surv.* 33.2 (June 2001), pp. 209–271. ISSN: 0360-0300.

[41] K. Vora et al. "Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 507–522.

[42] K. Wang et al. "GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC". In: *USENIX ATC*. 2015.

[43] P. Wang et al. "Replication-Based Fault-Tolerance for Large-Scale Graph Processing". In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. 2014, pp. 562–573.

[44] Y. Weiss and W. T. Freeman. "On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs". In: *Information Theory, IEEE Transactions on* 47.2 (2001), pp. 736–744.

[45] Yahoo! "altavista web page hyperlink connectivity graph". In: *circa 2002*. 2012. URL: http://webscope.sandbox.yahoo.com/.

[46] F. Yang and A. A. Chien. "Understanding Graph Computation Behavior to Enable Robust Benchmarking". In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '15. Portland, Oregon, USA: ACM, 2015, pp. 173–178.

[47] J. Yang and J. Leskovec. "Defining and Evaluating Network Communities Based on Ground-truth". In: *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. MDS '12. Beijing, China: ACM, 2012, 3:1–3:8.

[48] J. Yang and J. Leskovec. "Defining and Evaluating Network Communities Based on Ground-truth". In: *Knowl. Inf. Syst.* 42.1 (Jan. 2015), pp. 181–213. ISSN: 0219-1377.

[49] L. Yuan et al. "Modeling the locality in graph traversals". In: *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE. 2012, pp. 138–147.

[50] L. Yuan et al. "Modeling the Locality in Graph Traversals". In: *ICPP*. 2012.

[51] P. Yuan et al. "Fast Iterative Graph Computation: A Path Centric Approach". In: *SC*. 2014.

[52] D. Zheng et al. "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 45–58.

[53] X. Zhu et al. "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning". In: *USENIX ATC*. 2015.

[54] X. Zhu et al. "Gemini: A computation-centric distributed graph processing system". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*. 2016.