GRAPHZ: FASTER GRAPH COMPUTING FOR COMMODITY PCS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ZHIXUAN ZHOU

CHICAGO, ILLINOIS
SPRING, 2015
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... v
LIST OF TABLES ......................................................... vi
ACKNOWLEDGMENTS ........................................................ vii
ABSTRACT ................................................................. viii

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

2 BACKGROUND AND RELATED WORK .................................. 4
  2.1 Computation Models ................................................. 4
  2.2 Distributed Graph Engines ......................................... 5
  2.3 Single Machine Graph Engines ..................................... 6

3 DEGREE-ORDERED STORAGE ............................................. 9
  3.1 The Vertex Label-space ............................................. 9
  3.2 The Vertex Index .................................................. 10
  3.3 Example .......................................................... 11

4 PROGRAMMING IN GRAPHZ ............................................ 15
  4.1 User-defined Datatypes ............................................. 15
  4.2 User-defined Functions ............................................. 17
  4.3 Execution Model .................................................. 18
  4.4 BFS Example ....................................................... 20
  4.5 Expressiveness Compared to GraphChi ............................... 22

5 IMPLEMENTATION ....................................................... 25
  5.1 Sio & Dispatcher .................................................. 26
  5.2 Worker & Semi-Message Model .................................... 27
  5.3 MsgManager & Semi-Message Model ................................. 28

6 EXPERIMENTS .......................................................... 30
  6.1 Experimental Setup ................................................ 30
    6.1.1 Hardware Platform .......................................... 30
    6.1.2 Inputs ....................................................... 30
    6.1.3 Graph Algorithms .......................................... 31
  6.2 Large Graph Performance .......................................... 32
  6.3 Medium Graph Performance ........................................ 33
  6.4 Small Graph Performance ......................................... 34
  6.5 Bulk vs. Asynchronous Execution ................................ 35
  6.6 IO Statistics ...................................................... 36
LIST OF FIGURES

2.1 Comparisons between BSP model and asynchronous model .................. 5
3.1 Example of Relabeling and tight ID slots ..................................... 12
4.1 Graph Processing in the View of Developers .................................. 19
5.1 Main Parts in GraphZ ................................................................. 25
6.1 Run time comparison on the large graph ........................................ 32
6.2 Run time comparison on the medium graph ..................................... 34
6.3 Run time comparison on the small graph ....................................... 35
6.4 Total amount of I/Os for medium graph ........................................ 37
6.5 Total amount of I/Os for large graph .......................................... 38
LIST OF TABLES

2.1 Qualitative Comparison of Graph Packages. . . . . . . . . . . . . . . . . . . 8
3.1 Example Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.2 Relabeling of Example Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.3 Edge List Stored on External Disk . . . . . . . . . . . . . . . . . . . . . . . . 13
3.4 ids_table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.5 id_offset_table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6.1 Iterations for Convergence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
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

Supporting large scale graph analytics has become a central problem for computer systems researchers. Recent advances have made it possible to perform large scale graph analytics on commodity PCs. Despite this progress, however, graph processing on a PC still suffers from the slow speed of backing stores compared to CPUs. This paper presents two optimizations that reduce memory footprint and disk accesses for graph processing. The first is degree-ordered storage, a new storage format that greatly reduces book-keeping overhead. The second replaces existing static messages with a new semi-message model where messages to in-memory vertices are processed immediately, decreasing IO pressure and increasing parallelism while maintaining consistent execution order. This semi-message mode, thus, provides further reduction in memory footprint and disk access. We implement these optimizations in a new graph analytics framework called GraphZ, which we release as open source. Our experimental results – using three different graph sizes and six different benchmarks – indicate that GraphZ is competitive with state-of-the-art solutions for small graphs. For large graphs (greater than memory size), GraphZ provides performance improvements in the range of 1.2-17× over existing state-of-the-art solutions.
CHAPTER 1
INTRODUCTION

Many important data sets are naturally structured as graphs, including the web and various social networks. Consequently a large number of frameworks have been proposed for analyzing and mining extremely large scale graphs. Many approaches support graph analytics on large-scale distributed computing systems [1, 2, 9, 12, 17, 19, 20, 23, 25]. Not all graph analysts, however, have access to such computing infrastructure. Thus, recent efforts have focused on making graph processing possible on a single, commodity PC [11, 14, 22].

On small systems, large graphs have to be processed out-of-core, meaning the disk becomes a fundamental bottleneck. GraphChi [14] 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 [19] and GraphLab [17]. In the vertex-centric model, users specify an update function and the graph engine repeatedly applies this update to each vertex. If an update needs to change the value of an adjacent vertex, it can send a message along one of its edges. In this edge-based message model, data is explicitly stored in graph edges to be read during subsequent updates.

While this programming model is popular, it can result in a large number of accesses to the backing store. And the model makes it hard to overlap computation and disk accesses. To overcome these performance bottlenecks, the X-Stream system uses an edge-centric model designed to maximize sequential store accesses and overlap computation and external storage access [22]. In the edge-centric model, data associated with edges is updated sequentially so that the edges can stream sequentially from the backing store. The edge-centric model is designed to reduce random accesses and increase the bandwidth to the store. It does not, however, reduce the total number of storage accesses, and it requires the use of a less intuitive programming model.

This paper introduces two new optimizations designed to greatly improve the performance
of vertex-centric big graph processing on small machines. The first optimization is a new storage format for vertices we call degree-ordered storage. This format greatly reduces the size of book-keeping data structures necessary for out-of-core graph processing, thus reducing storage requirements and increasing the proportion of a graph which can fit in memory.

The second optimization replaces edge-based messages with a new semi-message model. Semi-messages are inspired by active messages [7] as they do not just move data, but also specify a function that can explicitly update a destination vertex. The semi-message model allows the graph processing runtime to intercept messages destined for in-memory vertices and update them immediately. Semi-messages reduce both memory footprint and storage accesses compared to passive edge-based messages. Both these optimizations save memory space and reduce the total number of IO operations.

We build both of these optimizations into a new graph analytics framework called GraphZ. GraphZ is implemented in C++ for Linux. We evaluate its performance using six different popular graph benchmarks on a desktop system with different amounts of RAM, different backing stores (HDD and SSD), and three different graph sizes. We compare GraphZ’s performance to both GraphChi and X-Stream, each state-of-the-art out-of-core graph engines.

For a small graph (LiveJournal [31]: 4M vertices, 69M edges), we find that there is no clear best solution – the best framework varies from application to application. For a medium graph (Friendster [31]: 66M vertices, 3.6B edges), we find that GraphZ’s optimizations provide significant advantages – 3× faster than GraphChi on average and as high as 16×; 12× faster than X-Stream on average and as high as 49×. Finally for large graphs (YahooWeb [29]: 720M vertices, 6B edges), GraphZ’s advantages hold – 3× faster than GraphChi on average and as high as 8×, and 3× faster than X-Stream on average with as high as 11×.

This paper makes the following contributions:

- It presents the degree-ordered storage format that reduces memory footprint and disk accesses for large scale graphs that must be processed out-of-core.
• It designs the **semi-message model** that further reduces memory footprint by replacing static messages that store data with messages that can actively perform computation on their recipient, further reducing memory footprint and disk accesses, yet maintaining the high efficiency of GraphChi’s **asynchronous execution model** [6, 14, 27].

• It implements these optimizations in GraphZ and performs an empirical comparison between GraphZ and two state-of-the-art out-of-core graph processing packages: GraphChi and X-Stream.

• It releases the GraphZ source code as open source \(^1\)

The rest of this paper is organized as follows. Chapter 2 presents background and related work. Chapter 3 describes the degree-ordered storage format. Chapter 4 describes the GraphZ programming model and compares it to GraphChi and X-Stream. Chapter 5 describes the implementation of the semi-message model. Chapter 6 evaluates GraphZ’s performance and compares it to GraphChi and X-Stream. Chapter 7 concludes.

---

1. https://github.com/danden89/GraphZ.
CHAPTER 2
BACKGROUND AND RELATED WORK

Many important data sets can be encoded as graphs. But graphs becomes bigger and bigger in this era.[13, 29] We can no longer easily load those big graphs fully into memory. And it’s hard to optimize graph processing because of the unstructured property of graphs. So researchers have found several ways to solve it: 1) Use clusters to provide enough RAM[9, 17, 19]; 2) Use SSD as complement for RAM[3]; 3) Do partitioning and use external disks [11, 14, 22].

To handle large graphs, a common method is to adopt the vertex-centric model, that a graph engines will iterate over vertices in the graph, and update their value with an update function defined by users. And under vertex-centric model, we have two most basic computation models: bulk-synchronous processing(BSP) model and asynchronous model.

To understand performance of different graph engines, Han has shown that work-balance, data locality, using asynchronous model, communication overheads, and memory efficiency are the key issues[10]. To get fair evaluations of graph engines, Fan has shown that experiments over a single algorithm or graph may produce unfair results and a selective set of algorithms could produce better evaluations[30].

In this Chapter, we’ll first compare the asynchronous model and BSP model. Then we’ll introduce state-of-art graph analytics frameworks for clusters and single machine.

2.1 Computation Models

For those first graph engines, they’re often implemented with BSP model, but more of them support asynchronous model now. The most basic difference between BSP model and asynchronous model is that vertices cannot see another vertex’s newest value in BSP model, while they can in asynchronous model. In general, asynchronous model performs better than
BSP model in converge speed and could be applied to some problems that couldn’t be solved by BSP model. But asynchronous model is often hard to get parallelized.

Here we give an example of how they behavior different on a BFS search. As Fig. 2.1 shows, we give each vertex a value by its id. And we choose vertex with id 0 as the root. For BSP model, we can search vertex 1 and 2, and assign them the value 0 of root. And after 2 iterations, we found all vertices. But in the asynchronous model, we can search all 3 vertices in a single iteration. Since that, after vertex with id 1 was founded, vertex 1 can immediately use its new value 0 to update vertex with id 3, then we can found them all. From this, we could see that, asynchronous model is more efficient than BSP model in general cases. So GraphZ choose this model to save iterations and disk access.

![Image](image1)

(a) BFS on BSP model

![Image](image2)

(b) BFS on asynchronous model

Figure 2.1: Comparisons between BSP model and asynchronous model

2.2 Distributed Graph Engines

A number of graph analytics frameworks have been developed. Many of these are designed to bring parallel computing resources to bear on large scale graphs. Giraph makes graph
processing possible on top of Hadoop [1]. Since Hadoop adopted the Map-Reduce, it cannot efficiently handle data dependencies and could incurs unnecessary disk accesses. Also, Giraph adopted the BSP model, so its processing speed is not that fast.

Pregel adopted a different approach, building a customized, distributed graph-processing engine [19]. It’s developed by Google and reported to work well. It also adopted BSP model, since BSP model is much easier to be parallelized in a distributed environment. Other distributed approaches followed including GraphLab [17], PowerGraph [9], and Hama [2]. Unlike previous engines, GraphLab adopted the asynchronous model and provide several different kinds of consistency model. And experiments shows that GraphLab runs way more faster than Giraph and converge faster than Pregel[10, 17].

Studies show that these custom frameworks provide better performance than building graph processing on top of existing infrastructures like MapReduce and Hadoop [10]. Subsequent work improves performance of these custom systems, especially through better distributed load balancing [12, 23], combining synchronous and asynchronous execution models [20], increasing graph processing fault tolerance [25] and providing online graph processing[28].

2.3 Single Machine Graph Engines

There are many challenges on implementing parallel graph processing [18]. One of the largest is accessing a large-scale distributed system in the first place. A personal user also may not have easy access to cluster. And Clusters are not easy to use that they can often have failures: disk failures, node failures, network congestion, etc. Fault-tolerance on them may not always work as it was supposed to be.

GraphChi makes this unnecessary by supporting large scale graph processing on a single, commodity PC [14]. GraphChi uses a vertex-centric programming model and an edge-based messaging model. To accommodate large graphs in small memories, it implements a parallel
sliding window that partitions the graph into pieces that fit in memory. The graph processing engine automatically orchestrates data movement from disk to memory. GraphChi’s execution model is asynchronous [6], in that it always performs computation using the most recent vertex data. Compared to approaches that use a bulk-synchronous model, where all vertices are updated at once, the asynchronous model results in fewer iterations [14]. The major drawback of GraphChi is that its model results in challenges on overlapping between computation and I/O, and thus achieve low utilization of I/O bandwidth.

To address the performance penalty of weak overlapping, X-Stream uses an edge-centric model [22]. This model is based on the observation that edges outnumber vertices, so edge updates should be made as fast as possible. Therefore, X-Stream is structured so that all edge updates can be done sequentially, allowing edge data to stream from the backing store in storage order. This streaming pattern greatly reduces random accesses, resulting in higher effective bandwidth and greater performance on a per iteration basis. This design, however, only works well under the bulk synchronous model, which can result in more total iterations [6]. In addition, using X-Stream requires abandoning the intuitive vertex-centric model of Pregel and GraphChi.

TurboGraph is a third approach supporting graph processing on a single machine [11]. Unlike GraphChi and X-Stream, TurboGraph is explicitly designed for high-end PCs with SSDs as backing stores. To fully utilize the fast random speed of SSDs, TurboGraph divides graphs into small 1MB blocks and uses a block list to search them. With large memory, it uses a pin-and-slide model to load only required blocks into memory to avoid unnecessary I/O operations. We also note that – unlike GraphChi, X-Stream, and GraphZ – TurboGraph is not open-source, meaning it is hard to compare to this approach empirically.

The goal of this paper is to further improve the performance of graph analytics on commodity PCs. The qualitative comparison of GraphZ and the most closely related work (GraphChi and X-Stream) is shown in Table 2.3. The table compares the programming,
Table 2.1: Qualitative Comparison of Graph Packages.

<table>
<thead>
<tr>
<th>Model</th>
<th>GraphChi</th>
<th>X-Stream</th>
<th>GraphZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>programming</td>
<td>vertex-centric</td>
<td>edge-centric</td>
<td>vertex-centric</td>
</tr>
<tr>
<td>execution</td>
<td>asynchronous</td>
<td>bulk-synchronous</td>
<td>asynchronous</td>
</tr>
<tr>
<td>storage</td>
<td>prefix-sum[8]</td>
<td>Edge List</td>
<td>degree-ordered</td>
</tr>
<tr>
<td>messaging</td>
<td>passive</td>
<td>passive</td>
<td>semi-messaging</td>
</tr>
</tbody>
</table>

execution, messaging, and storage models of these three approaches. As can be seen in the table, the primary contribution of GraphZ is to augment the vertex-centric, asynchronous programming model of GraphChi with novel storage and messaging models. These changes greatly reduce the IO burden of out-of-core graph processing and provide large speedups compared to GraphChi.
CHAPTER 3

DEGREE-ORDERED STORAGE

This storage format greatly reduces the space required to hold the graph vertices. For out-of-core processing, this optimization increases the number of vertices that can fit into memory at one time. As we fit more vertices into memory, we also increase the chance that a message’s destination will be in memory at the same time as the source, which maximizes the advantages of GraphZ’s semi-message model.

To reduce the memory impact of vertices, degree-ordered storage compresses two key data structures. The first is the vertex namespace, or label-space, and the second is the vertex index. The vertex label-space is the set of allowable identifiers for vertices. The vertex index is essential for partitioning graphs for out-of-core processing, as it is the data structure that tracks each vertex’s location on the file system.

This section describes the proposed solutions for compacting these key data structures. It then presents a small example to demonstrate their use in practice.

3.1 The Vertex Label-space

Each vertex in a graph has an identifier, or ID. We use the term vertex label-space to refer to the set of allowable IDs, typically integers. Our motivation to compress these IDs comes from the observation that many available datasets have a largest vertex ID that is much larger than the actual number of vertices. For example, the YahooWeb graph [29] has only about 0.7 B effective vertices. If you looked only at the largest ID, however, you would think it has about 1.4 B vertices. About 0.7 B ID slots are either empty or occupied by isolated vertices. Most graph analytics software allocates storage for every possible entry in the vertex label-space. If not all of the labels are actually used, then this allocation policy wastes precious memory resources.
For example, suppose that every vertex requires 8B of storage. Allocating for every entry in the label-space requires $8 \times 1.4 = 11.2$GB of storage for YahooWeb. If we compress the label space to account only for present vertices, we need $8 \times 0.7 = 5.6$GB. Using common out-of-core partitioning schemes like those proposed in GraphChi and X-Stream, compressing the label space would allow us to cut the number of partitions in half, greatly reducing disk access. Furthermore, this compression increases the chances that a vertex will be in memory when another vertex sends it a message, taking maximum advantage of GraphZ’s semi-message model.

Of course, graph analysts will want to refer to the original labels (which presumably have some intrinsic meaning) rather than the compressed label-space used by GraphZ. Therefore, GraphZ maintains the mapping relationship between a vertex’s new ID and old ID on external storage. GraphZ does not need to keep this relationship in memory, as it does not refer to it during the computation. Instead, it does a few lookups before a computation begins and gathers results after the computation ends.

### 3.2 The Vertex Index

The vertex index is a key data structure for out-of-core graph processing, as it keeps track of where all vertices are located. This mapping of vertex to location allows the large graph to be partitioned into small sets of vertices which fit into memory. While many graph packages use the *compressed sparse rows* format [24] for storing graphs, we introduce a new format: *degree-ordered storage*.

In the degree-ordered format, we sort vertices by out-degree. Then, we give each vertex a new ID based on this sorted order and update all the adjacency lists accordingly. This simple process both compresses the vertex label-space and reduces the index size compared to the compressed sparse rows format.

Rather than store an index for every vertex, the degree-ordered format simply stores the
smallest id of the group of vertices that have the same out-degree. The number of different out-degrees tends to be very small in natural graphs [9], so this format requires a (typically) much smaller number of indices than vertices.

For the YahooWeb graph, the number of different degrees is less than 100k. The degree-ordered format uses less than $100 \times 16 = 1.6$ MB to hold the vertex index. But if we want to store them in prefix-sum or CSR (without compacting the label-space), we need about $1.4 \times 8 = 11.2$ GB. This tight format (almost an order of magnitude reduction over common techniques) makes it possible to store the entire index in memory at one time, greatly reducing the time to search 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.

3.3 Example

This example illustrates how to convert a graph to degree-ordered format. We use the graph illustrated in Figure 3.1(a). Table 3.1 shows the adjacency list for this graph. We note that the maximum ID in the original graph is 12, although there are only 7 vertices. Thus this graph illustrates how the proposed processes will compress both the label-space and the index.

The first step in the process is to sort the vertices in order of ascending out-degree, with ties broken randomly. Table 3.2 shows a sorted order in columns 1–3. The next step is to relabel the vertices based on this order. This relabeling is shown in columns 4–5 of the table. The relabeled graph is shown in Figure 3.1(b).

Having relabeled the graph, we now need to store the following data: the map between the old and new IDs (columns 1 & 4 from Table 3.2), the ordering on the adjacency lists (column 5 from Table 3.2, as illustrated in Table 3.3), a mapping from degree to the first ID having this degree, and mapping from degree to the first out-neighbor's offset of first ID.
Figure 3.1: Example of Relabeling and tight ID slots

<table>
<thead>
<tr>
<th>source</th>
<th>dest</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2, 4, 12</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2, 7</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2, 10</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 10</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Relabeling of Example Graph

<table>
<thead>
<tr>
<th>source</th>
<th>dest</th>
<th>degree</th>
<th>new source id</th>
<th>new dest id</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2, 7</td>
<td>2</td>
<td>3</td>
<td>0, 6</td>
</tr>
<tr>
<td>4</td>
<td>2, 10</td>
<td>2</td>
<td>4</td>
<td>0, 1</td>
</tr>
<tr>
<td>0</td>
<td>2, 4, 12</td>
<td>3</td>
<td>5</td>
<td>0, 2, 4</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 10</td>
<td>3</td>
<td>6</td>
<td>0, 1, 3</td>
</tr>
</tbody>
</table>
Table 3.3: Edge List Stored on External Disk

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

Table 3.4 is the lookup table mapping degree number to the first ID having this degree. We call this \textit{ids\_table}. Instead of storing and index for every vertex, this table stores the smallest ID of the vertices with the same out-degree.

Table 3.4: ids\_table

<table>
<thead>
<tr>
<th>degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.5 is the lookup table for mapping from degree number to the offset of first id having this degree. We call it \textit{id\_offset\_table}. Combined with the \textit{ids\_table}, we also store the edges starting offset of the smallest vertex. Then a simple calculation tells us how many bytes to read for this vertex. This storage format is trading increased computation (for computing indices) for decreased memory footprint. For out-of-core graph processing this is an easy tradeoff to make – memory is a much more precious resource than computation.

Table 3.5: id\_offset\_table

<table>
<thead>
<tr>
<th>degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Also, this storage format is very good for random access, we only need to keep \textit{ids\_table} and \textit{id\_offset\_table} in memory. Since the the graphs are often sparse, these two tables typically take just hundreds of Kilobytes. To randomly access a vertex \( x \), we do a binary search on \textit{ids\_table} to find the degree \( d \) satisfying \( \textit{ids\_table}[d] \leq x < \textit{ids\_table}[d+1] \). This
tells us the out-degree of $x$ and the first id $ids_table[d]$ that has out-degree of $d$. Then we lookup the $id_offset_table$ to get the offset of vertex $ids_table[d]$. Finally we can easily compute the offset of vertex $x$, by the formula:

$$
offset = id_{-}offset_{-}table[ids_{-}table[d]] + (x - ids_{-}table[d]) \times d
$$ (3.1)

For example, if we want to find the offset of vertex 4, first we do a binary search on $ids_{-}table$, and find the degree of vertex 4 should be 2, and the first vertex having out-degree 2 is 3. Then we check the $id_{-}offset_{-}table[2]$, and get the offset of vertex 3 is 2. Based on the degree is 2, we can get the offset of vertex 4 is $2 + (4 - 3) \times 2 = 4$. And since the degree is 2, we need to read two edges in the disk file of offset 4 and 5. Then we read the disk, and get the out-edges, 0 and 1.
In this section we describe the process of writing GraphZ programs. The goal is to keep the vertex-centric programming model and asynchronous execution model that has worked well for GraphChi. We replace GraphChi’s passive messaging model with GraphZ’s novel semi-messaging. This section describes the way programmers interact with GraphChi.

GraphChi programmers define vertex and message data types. They then define two key functions: update() and apply_message(). The GraphChi runtime iterates through the vertices, calling the update function on each, which may create messages. If the GraphChi runtime detects a message’s destination vertex is in memory, it will call function apply_message() directly. This is the basis of the semi-message model.

This section details the GraphZ programming model. It first describes the data types and functions that users need to define, and then GraphChi’s execution model. We then present an example implementing BFS in GraphZ. Finally, we demonstrate that the GraphZ model is as expressive as GraphChi. Where appropriate we compare the GraphZ model to GraphChi and X-Stream.

### 4.1 User-defined Datatypes

All out-of-core graph engines are based on the concept of messages. The large number of vertices are partitioned into sets that fit into memory. If an update to one vertex causes an update to a neighbor which is not in memory, it is not feasible to load that neighbor from disk and process it immediately. Instead, a message is left and that vertex receives the message later when it is loaded from disk. Therefore, there are two main data types that need to be defined in a GraphZ program (shown in Algorithm 1): VertexDataType and MessageDataType.
MessageDataType is very similar to EdgeDataType in GraphChi. Unlike other graph processing engines, GraphZ does not give every edge in the graph a value. For many algorithms, we do not need edge values at all. If it is needed, we consider a vertex’s edges’ values as part of its vertex value, so we do not lose functionality compared to GraphChi. For more details, please refer to Sec. 4.5.

Users can define VertexDataType and MessageDataType to map to a number of different C++ datatypes. All atomic datatypes such as int, double, bool, etc. are applicable. Structs combining data types are also appropriate. For example, to implement the PageRank algorithm, the users could 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 one for possible value change.

**Algorithm 1** Key Data Structures

```
1: struct VertexDataType {
2:   int vval1;
3:   float vval2;
4:   double vval3;
5:   ..... // it could be int, float, struct, array....
6:     ▷ Value initiation could be done under constructor
7: }
8:
9: struct MessageDataType {
10:  int msg_val1;
11:  float msg_val2;
12:  double msg_val3;
13:  ..... // it could also be int, float, struct, array....
14: }
```

16
4.2 User-defined Functions

To develop new algorithms on GraphZ, users need to define two functions, `update()` and `apply_message()`. As Algorithm 2 shows, `update()` should address the following tasks: adjusting a vertex's value if needed, computing new messages, iterating over its out-neighbors, and deciding whether to send them messages.

The `apply_message()` routine defines the work to be accomplished when a vertex receives a message. When the GraphZ runtime intercepts a message whose destination is in memory, it will automatically call this function on the destination. In the example of 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

1: function UPDATE(vertex)
2:     vertex ← $f_1$ (vertex)  \(\triangleright\) not a must
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:     
8: function APPLY_MESSAGE(vertex, msg)
9:     vertex ← $f_2$ (vertex, msg)
10:    return vertex
11: \(\triangleright\) $f_2$ is often a very easy function, like $\min\text{\textit{(vertex, msg)}}$, $\text{\textit{vertex + msg}}$, $\text{\textit{vertex.append(msg)}}$

To be clear, for many algorithms, there is no need to adjust a vertex’s value $\text{\textit{vertex}}$ at the beginning of `update()`. Algorithm 2 just shows a reference process, developers could freely define the `update()` function that fits their needs.

Algorithm 3 shows the functions users must define in GraphChi. GraphChi requires only an `update()` function, which must: iterate over in-edges and read edge data, update the vertex’s value, compute a new edge-value, iterate over out-edges and decide whether to write the new messages to a specific out-edge.
Algorithm 3 User-defined Methods in GraphChi

1: function Update(vertex)
2:     for in_edge in vertex's in_edges do
3:         read edge data from in_edge
4:         if some condition then
5:             update vertex
6:             \(\triangleright\) this vertex update could also be done outside this for loop
7:     compute a message msg
8:     for out_edge in vertex's out_edges do
9:         if some condition then
10:            write msg to out_edge

Algorithm 4 User-defined methods in X-Stream

1: function Generate_Update(edge)
2:     get the source vertex value src of edge
3:     get a blank update from update_stream
4:     fill the update by computation
5: function Apply_Update(update)
6:     get the destination vertex value dest of update
7:     update dest with update

Unlike GraphZ and GraphChi, X-Stream uses the very different edge-centric model [22]. Where both GraphChi and GraphZ iterate over vertices, X-Stream iterates over edges generating and applying updates (see Algorithm 4). As X-Stream iterates over edges, critical information must be assembled by the developers. For example, at the first iteration of PageRank on X-Stream, the developer needs to count every vertex’s degree and cannot do real computation. That computation can only be performed on subsequent rounds after the degree has been determined. This edge-centric model may put a greater burden on developers.

4.3 Execution Model

To use GraphZ, developers should not only define the required data types and functions, they must understand how the GraphZ engine will apply those functions. GraphZ uses the same asynchronous execution model as GraphChi. This execution model is illustrated in
The GraphZ engine manages two lists: `vertices_array` and `vertices_adjacents_list`. Here, `vertices_array` stores the values of all vertices and `vertices_adjacents_list` stores the adjacency list for every vertex in degree-sorted order. GraphZ only stores vertices’ out-neighbors — ones to whom there exists an edge from the vertex. Most algorithms do not need to know the in-neighbors of a vertex. If required, in-neighbors can be added to messages.

The GraphZ engine iterates through `vertices_array` (which is stored from lowest to highest degree). The method `update()` is called on every vertex. Fig. 4.1, shows arrows pointing two different directions. The upward arrows indicate that when the GraphZ engine is updating a vertex, it may (depending on the `update()` method) read that vertex’s value and combine it with the adjacency list to adjust vertex’s value and make decisions on messages. Downward arrows represent sending messages to a vertex’s out-neighbors. These messages are handled automatically by the GraphZ engine, which calls the `apply_message()` routine.
GraphZ is inherently iterative. 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.

Though GraphZ is a multi-threaded graph engine, it still ensures a strong consistency guarantee, which allows users to reason about GraphZ programs as if they were executed by a single-threaded, sequential program. Specifically, for any 2 vertices \( v_1 \) and \( v_2 \), if \( v_1 < v_2 \), then \( v_1 \)'s value would always be updated before \( v_2 \) in every iteration and messages sent during the update execution of \( v_1 \) would also be applied before updating \( v_2 \). GraphChi also maintains such a strong consistency, and we believe this guarantee makes it easy to develop and test new algorithms. Besides ease-of-use, maintaining consistency also has a performance advantage, which can greatly accelerate the converge speed and reduce disk access.

### 4.4 BFS Example

We illustrate GraphZ programming using breadth first search as an example. At the start of computation, we initialize all vertex values to \(+\infty\) except the root vertex. We consider vertex with id 0 as the root. At each iteration, if a vertex’s value was updated, then it sends messages to its neighbors, so that other vertices could also be found. After the process converges (no vertex’s value was updated in the final round), we can check those vertex’s values and if they are changed to 0, we consider them found.

First, as shown in Algorithm 5, we define \( \text{VertexDataType} \) as a struct of two \( \text{unsigned int} \)s \((vval \text{ and new } vval)\) and we define \( \text{MessageDataType} \) as \( \text{unsigned int} \). Here we define two values for the vertex, to track whether the vertex was found and also whether the vertex’s value was changed since the last \( \text{update}() \). If the value was not changed, we do not send any messages in the current \( \text{update}() \).
Algorithm 5 BFS Data Structures

1: struct VertexDataType {
2:   unsigned int vval;
3:   unsigned int new_vval;
4: }
5:
6: struct MessageDataType {
7:   unsigned int msg;
8: }

Algorithm 6 BFS

1: function UPDATE(vertex)
2:   if cur_iter == 0 then
3:     if vertex.id == 0 then
4:       vertex.vval ← 0
5:       vertex.new_vval ← 0
6:     else
7:       vertex.vval ← +∞
8:       vertex.new_vval ← +∞
9:     return
10:   if vertex.vval ≤ vertex.new_vval then return
11:   vertex.vval ← vertex.new_vval
12:   for vadj in vertex’s adjacent vertices do
13:     send message vertex.vval to vadj
14: function apply_message (vertex, msg)
15:   if msg ≤ vertex.new_vval then
16:     vertex.new_vval ← msg
17:     return vertex
As Algorithm 6 shows, the `update()` method first checks the current iteration number. If it is the first iteration, we do some initialization setting vertex with id 0 to 0 and other vertices to $+\infty$. If the current iteration is not the first, then we need to check whether $vval$ and $new\_vval$ are the same. If they are the same, then we know that this vertex’s value was not changed in the last `update()`, and we can directly return without sending any more messages to its out-neighbors.

This method saves many I/O accesses, since messages across partitions have to be stored but messages to current partition could be applied immediately. In GraphChi, however, the GraphChi engine will always store those changed edges whether they belong to the current working partition or not. GraphChi manages edge data in blocks, that could be as large as 1MB, so even if we only change 1 edge’s data in the block, the full block needs to be written back to disk. And at next iteration, we always need to read the whole block back and check for messages. This will cause many reads and writes.

### 4.5 Expressiveness Compared to GraphChi

The GraphZ interface attempts to keep many of the best features of GraphChi, but it requires a slight modification to implement under the semi-message model. Namely, GraphZ does not store edge data, and it does not send messages along edges. Instead GraphZ eliminates edges storage and 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. In this section, we argue that these changes do not reduce expressiveness compared to GraphChi.

We prove this claim by showing how to convert a GraphChi program into a GraphZ program. As shown in Algorithm 7, 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` is corresponding to the edge value in GraphChi. As noted before, we add `edges`
to `VertexDataType`, thus a vertex's edges is part of the vertex. `RealVertexDataType` is the real value we want to have for the vertex.

**Algorithm 7** Data Structures for Emulating GraphChi

```c
1: struct Edge {
2:     vertex_id neighbor;
3:     EdgeDataType edge_val;
4: }
5:
6: struct VertexDataType {
7:     List <Edge> edges;
8:     RealVertexDataType vertex_val;
9: }
10:
11: struct MessageDataType {
12:     Edge edge;
13: }
```

In Algorithm 8, we then make `update()` function more like GraphChi. We consider the variable `edges` exactly as those in-edges in GraphChi. And here, developers can read them all in and update the real `vertex_val`. Then we do the same work of 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.

Finally, though we have a constructive proof that GraphZ is as expressive as GraphChi, that does not mean GraphZ is always more efficient after such a conversion. GraphChi is designed to use edges as intermediate storage for messages and has some optimization for storing such static data types. GraphZ, in contrast has to store dynamic data for those conversions, so we do not expect GraphZ programs constructed in this manner to achieve much higher performance because of the very expensive storage for dynamic data.
Algorithm 8 Program Conversion

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: 16: function apply_message (vertex, msg)
17:     vertex.edges.append(msg.edge) return vertex

24
CHAPTER 5
IMPLEMENTATION

This section describes the implementation of GraphZ’s processing engine. These are internal details and do not directly affect the users, except in that the engine implementation is responsible for delivering performance.

As shown in Fig. 5.1, GraphZ’s implementation is composed of 4 major parts: Sio (Sequential I/O), Dispatcher, Worker, and MsgManager. The engine divides a large graph into partitions which fit into available memory. To iterate over a partition, the MsgManager loads all the vertices in the partition into memory then calls the user-defined `apply_message()` function to any destination vertices within the partition. Next, Sio reads the graph storage file into memory and reads edge blocks. These edge blocks are passed to the Dispatcher, which translates them into adjacency lists. The Worker block calls the `update()` function
on each vertex in the partition. During the process of iterating through the vertices, the Worker may intercept some messages and execute the `apply message()` function directly on any recipients that are in the current partition. For all other messages, they are passed to MsgManager. MsgManager will write them to disks. The Worker and MsgManager combine to implement the semi-message model. **The fact that messages to in-memory vertices are handled immediately is one of the keys to GraphZ performance.**

To synchronize these 4 components, we use the traditional fork-join model. Sio cannot start to retrieve blocks until the last partition was done. When the graph is small and the disk is very fast, it has have some observable overheads on the starting overhead of each partition. But when the graph is large, we can ignore the negative influence. GraphZ focus on out-of-core large graph processing, so we leave these corner-case optimizations to future work.

To communicate with each other, those 4 major 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. Our attempts to implement lock-free queues have resulted in worse performance, so we use traditional locking schemes.

Finally, all data transferred between the 4 main components are organized in small groups, which enables the engine 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 overlapping between computation and I/O.

### 5.1 Sio & Dispatcher

Sio is short for Sequential I/O retriever. At the start of iterating over every partition, Sio will load graph description files (like Table 3.4 and Table 3.5) into memory. Since they are very small, it is affordable to load them fully into memory. Since those tables are naturally sorted on the sequence of ids, Sio just needs to read it from 0. GraphZ does not manage
those data blocks by itself. Since it always reads data in a sequential manner, it simply relies on the OS to do prefetching and caching.

To exploit the maximum IO bandwidth of disks and get better pipelining, we split the adjacency list construction work to the Dispatcher. The Dispatcher receives blocks from Sio and parses the block into several adjacency lists. For example, Sio will tell the Dispatcher the length $len$ of a block, the startId is $l$, the endId is $r$. 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 will allocate memory for each vertex’s adjacency list.

### 5.2 Worker & Semi-Message Model

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

GraphZ replaces the edge values with messages, yet keeps the high efficiency of the asynchronous model. As noted in last paragraph in Sec. 4.3, GraphZ needs to maintain the execution order. So we add the message intercepting mechanism to GraphZ, to ensure the execution order. When an invocation of the `update()` method sends a message, the Worker intercepts it and determines if this message should be applied immediately or forwarded to the MsgManager.

---

**Algorithm 9** Message Intercepting (Part a.)

```
1: procedure MAIN WORKER
2:   for vertex in current partition do
3:     do update
4: 
5: function SEND_MESSAGE(as)
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. As shown in Algorithm 9, the
Worker iterates over vertices applying the `update()` function defined by developers. When executing `update()`, it will call the `send_message()` function. `send_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, we could see that message with destinations on current partition would be updated immediately, so we won’t violate 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.

### 5.3 MsgManager & Semi-Message Model

The MsgManager may be the busiest component in GraphZ. It needs to: 1) Wait for messages, write to the right partition and flush full buffers to disk and 2) Before an iteration on a partition starts, load all vertices of the current partition into memory, read all messages for this partition, and apply them to vertices of this partition.

Algorithm 10 shows the first task. The MsgManager always waits for messages and puts the message into the right buffer. For every partition, there is a corresponding buffer, so that the MsgManager could read them back and apply them to the right partition.

```
Algorithm 10 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 to iterate over a new partition, the MsgManager needs to first flush vertex values of the last active partition back to disk and load 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. This parallelization does not violate our consistency guarantee, since all vertices in the next active partition have not yet entered the update(). To avoid possible conflicts when multithreading may apply the same vertex, we use a mutex pool to solve conflicts. 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.
CHAPTER 6
EXPERIMENTS

In this section we evaluate the optimizations embodied in GraphZ by comparing it to GraphChi and X-Stream. We begin by describing the system, benchmarks, and inputs used in this study. We then describe the performance for the large, medium, and small graphs. Next we look at the benefits of the asynchronous model. We then compare the IO burden of these three approaches.

6.1 Experimental Setup

6.1.1 Hardware Platform

We use a machine with an Intel i7 2700K (4 cores, 8 hardware threads), and different sizes of RAM (4GB, 8GB, 16GB). The system runs CentOS 7 and Linux kernel 3.10.0. The machine has 3 disks, an internal 250GB mechanical disk, an internal 500GB Samsung 840 Pro SSD and an external 4TB mechanical disk connected with an eSATA cable. We use the 250GB disk for OS, and do experiments on the other two. To make the experiments more accurate, we configure the machine without any swap partitions, so that we can eliminate the possible interference of system memory replacement.

6.1.2 Inputs

For datasets, we use 3 graphs, LiveJournal [31], Friendster [31], and YahooWeb [29]. The LiveJournal graph has about 4M vertices, and 69M edges. The Friendster graph has about 65.6M vertices and 3.6B edges. The YahooWeb graph is the largest graph to which we have access, with 720M vertices and 6B edges, whose largest vertex ID is about 1.4B. As mentioned in Sec. 3, the largest vertex ID does not mean the graph has such number of vertices.
The YahooWeb graph requires more than 70 iterations to converge in BFS, CC and SSSP. To get more timely results, we set all 3 engines to run for fixed 5 iterations on those 3 algorithms. This may overestimate X-Stream’s performance as its bulk-synchronous model typically requires more iterations than the asynchronous models used by GraphChi and GraphZ.

### 6.1.3 Graph Algorithms

We do experiments on 6 graph algorithms, Connected Components (CC), Breadth-first search(BFS), PageRank (PR) [15, 21], Single-Source Shortest Path (SSSP), Belief Propagation (BP)[4, 26], and Randomwalk (RW) [16]. GraphChi comes with CC, Pagerank, and Randomwalk, while X-Stream lacks Randomwalk. We implement those missing algorithms ourselves. CC is a way to find connected components among vertices in graphs. Given graph \( G = (V, E) \), for any two vertices \( a, b \), if \( \exists \) a vertex \( c \), such that \( (a, c) \in E \) OR \( (c, a) \in E \) \( \text{AND} \) \( (b, c) \in E \) OR \( (c, b) \in E \), then we consider \( a \) and \( b \) are in the same connected-component.

For both BFS and SSSP, we define a starting root, then we start computations. BP is an algorithm that compute the sum-product of messages to get the margin distribution[4, 26]. In GraphZ and GraphChi we implement a modified version of BP which sends messages to its neighbors without eliminating messages received by them. It has similar performance as the original version on image denoising. PageRank is an algorithm used to recognize important nodes. By doing message passing and accumulating messages, a vertex with more directed/indirected incoming edges would be more important [15, 21]. In RandomWalk, we assume some humans on some vertices, and in every iteration, they can go anywhere with the same probability along out-edges. The more a vertex was walked over in current iteration, the more important it is [16]. PageRank, RandomWalk and BP are used to find important nodes.

For some combinations of benchmarks, storage, and memory sizes we are unable to obtain
results for GraphChi or X-Stream. We list those cases and the cause of the errors here. 1) Disk input/output error; 2) Disk is full; 3) Killed by Kernel; 4) Failed to alloc memory; 5) Hang. If we are unable to obtain a particular result, that entry in the charts is blank and we do not include that point for any graph processing engine when we compute aggregate statistics.

### 6.2 Large Graph Performance

![Figure 6.1: Run time comparison on the large graph.](image)

The results for the large graph (YahooWeb) are shown in Fig. 6.1. This figure has 12 charts, the left column shows the results for the magnetic disk. The right column shows results for the SSD. Each row of charts corresponds to a different benchmark. The $x$-axis
shows the amount memory used in the benchmark, while the $y$-axis shows the execution time (with lower times representing improved performance).

These results demonstrate that GraphZ achieves significantly lower run times than either GraphChi or X-Stream. On average for the HDD, GraphZ is about $5.5 \times$ faster than GraphChi and $3.1 \times$ faster than X-Stream. GraphZ’s maximum speedup is $7.7 \times$ compared to GraphChi on SSSP and $4.3 \times$ compared to X-Stream on SSSP.

On average for the SSD, GraphZ is about $2.5 \times$ faster than GraphChi and $3.8 \times$ faster than X-Stream. GraphZ’s maximum speedup is $4.2 \times$ compared to GraphChi on BFS and $11.4 \times$ compared to X-Stream on RW. All approaches benefit tremendously from moving to SSD. GraphZ still provides a significant performance gain, however.

None of the three approaches obtained a result for Randomwalk on the large graph with the HDD. All three appear to hang. We did obtain results for Randomwalk using X-Stream and GraphZ on the SSD, though.

### 6.3 Medium Graph Performance

The results for the medium graph (Friendster) are shown in Fig. 6.2. The layout of this figure is the same as that for the large data, except we show results only for the 4GB memory size. The different memory sizes produced almost identical results for this graph so the 8 and 16GB sizes are omitted for space.

The relative performance difference between GraphZ and the other packages is even larger in this case. On average, for the HDD, GraphZ runs about $8.7 \times$ faster than GraphChi and $7.9 \times$ faster than X-Stream. GraphZ’s maximum speedups are $16.7 \times$ compared to GraphChi on BFS and $15.3 \times$ compared to X-Stream on BFS.

On average, for the SSD, GraphZ is $3.7 \times$ faster than GraphChi and $15.7 \times$ faster than X-Stream. The maximum speedup is $7.5 \times$ for GraphZ compared to GraphChi on SSSP and $49 \times$ for GraphZ compared to X-Stream on RW.
The results for the small graph (LiveJournal) are shown in Fig. 6.3. The layout of this figure is the same as the previous two. In this case, the graph fits into the 4GB memory, so we only show results only for the 4GB memory size.

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 significantly change the results.

For small graphs like LiveJournal, optimizations for in-memory footprint 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. 6.3,
Figure 6.3: Run time comparison on the small graph. however, shows the results of in-memory graph processing with GraphZ are competitive or even sometimes much better than existing approaches.

### 6.5 Bulk vs. Asynchronous Execution

Graph processing is inherently iterative. Each framework (GraphChi, X-Stream, and GraphZ) continually iterates over the vertex space. GraphChi adopted an asynchronous execution model to reduce the total number of iterations compared to a 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 6.1 shows the number of iterations required for convergence for each approach for SSSP, CC, and BFS on both the LiveJournal and Friendster graphs. Since both GraphChi
Table 6.1: Iterations for Convergence

<table>
<thead>
<tr>
<th></th>
<th>SSSP</th>
<th>CC</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphChi</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>X-Stream</td>
<td>47</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>GraphZ</td>
<td>10</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Friendster</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphChi</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>X-Stream</td>
<td>59</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>GraphZ</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

and GraphZ use the asynchronous model, they require significantly fewer iterations to converge than X-Stream. This reduction in iterations is a key factor leading GraphZ to outperform X-stream on commodity platforms.

The results show that GraphZ uses a few more iterations than GraphChi, because GraphZ’s degree-order storage sorts vertices in ascending order. In every partition, vertices with smaller degrees are processed before those with larger degrees. This means that the vertices which send the most messages (as they have the highest out-degree) are processed last in each iteration. This data suggests that future GraphZ implementations should change the ascending order to descending. We save this optimization for future work.

6.6 IO Statistics

Throughout the paper, we argue that GraphZ’s degree-ordered storage and semi-message model will greatly reduce the IO burden. We have already seen that GraphZ provides much greater performance for large size graphs. Here we evaluate the IO operations explicitly.

Fig. 6.4 compares the exact external I/O of the three graph engines on two algorithms: PageRank and BFS with medium graph Friendster. These results are representative of the I/O statistics for all algorithms, the others are omitted for space. The figure shows that when running PageRank using 4GB RAM, GraphZ does only 1/3 of the reads of GraphChi,
Figure 6.4: Total amount of I/Os for medium graph
Figure 6.5: Total amount of I/Os for large graph
and just 1/4 that of X-Stream. Using 16GB RAM, GraphZ requires only 1/5 the reads of GraphChi and just 1/7 that of X-Stream. When running BFS with 4GB RAM, GraphZ requires only 1/3 the reads of GraphChi and 1/10 that of X-Stream. Using 16GB RAM, GraphZ needs just 1/20 the reads compared to GraphChi and 1/70 compared to X-Stream. Fig. 6.5 compares I/O on the large graph, and GraphZ still reduce a lot of I/O burden. This data confirms that GraphZ’s optimizations save a tremendous amount of IO compared to other state-of-the-art approaches.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This paper has presented two methods of improving large-scale graph analytics on small-scale systems. The first method is a new 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 semi-messaging model, 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.

We have combined degree-ordered storage with the semi-messaging model in the GraphZ framework. 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.

7.2 Future Work

Within the process of building GraphZ, we have shown the advantages of degree-ordered storage and semi-message model. But we have a few more questions to be answered. Is it worthy for us to adding the support for dynamic data types to expand GraphZ’s expressiveness? Can we utilize other computation methods like selective scheduling to accelerate the out-of-core graph processing? Is it worth to low down the network communication in distributed Graph engines? How close will GraphZ’s output be close to the ideal output when given a fixed energy budget?
First, for expressiveness, currently GraphZ supports those algorithms that only require static data types. For example, the original version of BP would require storing all messages, since messages cannot be aggregated. But adding support for dynamic data types may scarifies performance. We can store static values in a simple array with very few data format descriptions, but we may need to add much more description data when storing dynamic data. FlatBuffer could be a solution for dynamic data type support with small computation and I/O overhead[5]. Besides, GraphZ’s low utilization of CPU should allow machine to allocate more computation resources to compensate FlatBuffer’s decode/encode process, which could be easily overlapped with I/O. Also, GraphZ has shown that trading computation time for I/O results in greater performance for out-of-core, so we could expect using FlatBuffer would be ideal for performance. But we need more research to show whether this hypothesis stands and whether it is valuable to make this support.

Second, as noted in Sec. 6.5, different iteration orders may cause different convergence speeds. We should try to converge as fast as we can to achieve both low elapsed time and small I/O access. In general, we should iterate on vertices that have big (maybe only potential) impacts first, so that graph engines could spread changes at a maximum speed. But for current GraphZ’s model, we put as more vertices in memory as we can instead of putting a whole partition in memory. So we can only iterate vertices in the sequence of reading from disks. If we want to reschedule update order, we will have to do random access on disks, which would greatly slow down the engine. So we may need smarter ways on organizing them. And we need to find that whether we can find the ideal order, and see how far GraphZ is from it. Finally, we need to show whether it worth to be closer to such order.

Third, as we see from GraphZ, reducing disk access could provide great performance improvement. On distributed graph engines, network communications could be the bottleneck as well. Generally, communication over network in distributed graph engines could incurs high latency, congestion and high synchronization overhead. If we can effectively reduce and
reorganize communications, we think there is a high probability to gain high improvement as well. Here we can try optimizing parallel asynchronous models and find efficient message storing and scheduling algorithms. When a message was produced, we need to make smart decision between sending it out, storing it, applying it to vertices or aggregate it with another message. Also, we need to find an ideal to compress messages and how large a best message buffer would be.

Finally, as we got much bigger web graph and social network than ever, and more companies use graphs to find unveiled knowledge, more and more energy is consumed on graph computation. But if we have gotten a fixed energy budget, we will have to produce approximate results sometimes. Here we need to find how close can those approximate results be to ideal ones and what optimizations would achieve the closest approximate result. There are several dimensional decisions to make here: 1) BSP or asynchronous model; 2) Distributed cluster or single machine; 3) Out-of-core or in-memory processing; 4) Exploit maximum parallelism of machines or find the best appropriate parallelism.
BIBLIOGRAPHY


