2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00033

MultiLogVC: Efficient Out-of-Core Graph Processing Framework for Flash Storage

1st Kiran Kumar Matam*
Facebook Inc.
Menlo Park, USA
kiranmatam@fb.com

2nd Hanieh Hashemi
Electrical Engineering Department
University of Southern California
Los Angeles, USA
hashemis@usc.edu

3rd Murali Annavaram
Electrical Engineering Department
University of Southern California
Los Angeles, USA
annavara@usc.edu

***Abstract*—Graph analytics are at the heart of a broad range of applications such as drug discovery, page ranking, transportation systems, and recommendation models. When graph size exceeds the available memory size in a computing node, out-of-core graph processing is needed. For the widely used out-of-core graph processing systems, the graphs are stored and accessed from a long latency SSD storage, which becomes a significant performance bottleneck. To tackle this long latency this work exploits the key insight that that nearly all graph algorithms have a dynamically varying number of active vertices that must be processed in each iteration. However, existing graph processing frameworks, such as GraphChi, load the entire graph in each iteration even if a small fraction of the graph is active. This limitation is due to the structure of the graph storage used by these systems. In this work, we propose to use a compressed sparse row (CSR) based graph storage that is more amenable for selectively loading only a few active vertices in each iteration. However, CSR based graph processing suffers from random update propagation to many target vertices. To solve this challenge, we propose to use a multi-log update mechanism that logs updates separately, rather than directly update the active edges and vertices in a graph. The multi-log system maintains a separate log per each vertex interval (a group of vertices). This separation enables efficient processing of all updates bound to each vertex interval by just loading the corresponding log. Further, by logging all the updates associated with a vertex interval in one contiguous log this approach reduces read amplification since all the pages in the log will be processed in the next iteration without wasted page reads. Over the current state of the art out-of-core graph processing framework, our evaluation results show that the MultiLogVC framework improves performance by up to $17.84\times$, $1.19\times$, $1.65\times$, $1.38\times$, $3.15\times$, and $6.00\times$ for the widely used breadth-first search, pagerank, community detection, graph coloring, maximal independent set, and random-walk applications, respectively.**

***Index Terms*—out-of-core graph processing, graph analytics, SSD storage systems, log storage**

I. Introduction

Graph analytics are at the heart of a broad range of applications. The size of the graphs in many of these domains exceeds the capacity of main memory of typical computer systems. Hence, many out-of-core (also called external memory) graph processing systems have been proposed. These systems primarily operate on graphs by splitting the graph into chunks and operating on each chunk that fits in main memory. For instance, GraphChi [14] and many follow on research papers [28], [31], format graphs into shards, where each shard has a unique data organization that minimizes the number of random disk accesses needed to process one chunk of the graph at a time.

In terms of programming, many popular graph processing systems use vertex-centric programming paradigm. This computational paradigm uses bulk synchronous parallel (BSP) processing where each vertex is processed at most once during a single iteration (or superstep), which may generate new update messages to their connected vertices. Each vertex that received an update in one superstep may process those updates in the next superstep, which may again modify the vertex state or generate updates to another vertex, or even mutate the topology of the graph for the next superstep. The graph is thus iteratively processed over multiple supersteps until no further updates are generated or some other convergence critieria is met.

The repeated supersteps of execution in vertex-centric programming, however, cause a major data access hurdle for shard-based graph frameworks. We show later in this work that as supersteps progress, the number of active vertices that receive messages, and hence must process those messages, continuously shrinks. Shard-based graph frameworks are unable to limit their accesses to a limited number of active vertices. In each superstep, they still have to read whole shards into memory to access the active vertices that are embedded in these shards.

It is this observation that active vertex set shrinks with each superstep that this work exploits as a first step. Given that shard-based graph formats do not support accessing just the active vertex set, we rely on an efficient compressed sparse row (CSR) representation of the graph to achieve this goal. While CSR format allows efficient access to a subset of vertices and their associated edges, it has one drawback. Since CSR format can either place incoming or outgoing edges in contiguous locations but not both, the problem of random access traffic is one of the drawbacks of CSR format (more details in the next section). To resolve this challenge, we design a novel multi-log graph processing paradigm that logs all the outgoing messages generated from each superstep into a collection of logs, where each log is associated with an interval (or a group) of vertices. Logging the messages, as opposed to embedding the messages

*This work was done while the author was a graduate student at USC as part of the thesis research.

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

as metadata into each edge, decouples the message placement from the organization of graphs, thereby providing new options to remove the random access hurdle.

The last challenge tackled by this work is that when using log-based message processing, each superstep still has to read the edge lists from storage. As we show later in our analysis, many of the edge list pages accessed in SSD contain edges that are from inactive vertices. Hence, to improve edge list access efficiency, we optimize the multi-log design by logging the edge lists of the vertex that is likely to be active in the next superstep. We use a lightweight history-based prediction to determine whether a vertex will be active in the next superstep.

Compared to prior approaches that also rely on log-based graph processing [6], [11], our approach differs in two important ways. Most log-based graph processing frameworks use a single log and *merge* the messages bound to a single destination vertex to optimize log access latency. However, many important graph algorithms, such as community detection [24], graph coloring [9], maximal independent set [19] must process all the messages *individually*. Hence, approaches that merge messages do not permit the execution of many important classes of graph algorithms. However, preserving all the messages without any merge process leads to extremely large log size, thereby creating significant log processing overhead at the beginning of each superstep [11]. Inspired by these challenges, we design a novel multi-log paradigm that allows efficient access to smaller logs, based on the vertex interval that is currently being processed. By preserving the messages without merging, we enable all graph algorithms to be executed within our framework, thereby supporting the generality of GraphChi-based frameworks [2], [14], [28], [31]. At the same time, we use CSR formatted graphs augmented with multi-log structures to access the active vertex set, thereby drastically reducing the number of unwanted page accesses to the storage.

Our main contributions in this work are:

- We propose an efficient external memory graph analytics system, called MultiLogVC (multi-log with vertex-centric generality), which uses a combination of CSR graph format, and message logging to enable efficient processing of large graphs that do not fit in main memory. Unlike shard-based graph structures, CSR format enables accessing only the pages containing active vertices in a graph. This capability is a key requirement for efficient graph processing since the set of active vertices shrink significantly with each successive superstep. Hence, CSR format reduces read amplification.
- To efficiently access all the updates in a superstep, MultiLogVC partitions the graph into multiple vertex intervals for processing. All the outgoing messages are placed into multiple logs indexed by the destination vertex interval. All the updates generated by a vertex to other vertex intervals are stored in their corresponding log. When an interval of vertices is scheduled for processing, all the updates it needs are located in a single log. We use SSD's capability for providing parallel writes to multiple channels for concurrently handling multiple vertex interval logs with only small buffers on the host side.
- Even with message logging, MultiLogVC must still access the outgoing edge lists of each active vertex to send messages for the next superstep. Our analysis of edge list accesses showed that many page accesses contain edge lists of inactive vertices interspersed with active vertices. To further reduce the read amplification associated with edge lists, MultiLogVC logs the edge list of any potential active vertex that may be active in the next iteration so that this edge list can be read more efficiently from the log, rather than reading a whole page of unneeded edge lists.

II. Background and Motivation

A. Out-of-core graph processing

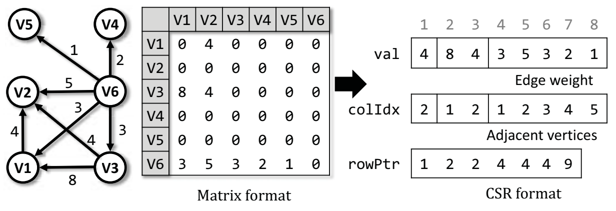
In the out-of-core graph processing context, graph sizes are considered large compared to the main memory size but can fit in the storage size of current SSDs (in Terabytes). GraphChi [14] is a representative out-of-core vertex-centric programming system and many further works built on top of this basic framework [2], [28], [31]. Thus, we will describe GraphChi's shard-based graph format and the associated challenges when processing with it.

GraphChi partitions the graph into several vertex intervals, and stores all the incoming edges to a vertex interval as a shard. Figure 1b shows the shard structure for an illustrative graph shown in Figure 1a. For instance, shard1 stores all the incoming edges of vertex interval $V1$, shard2 stores $V2$'s incoming edges, and shard3 stores incoming edges of all the vertices in the interval $V3-V6$. Also each shard stores all its in-edges sorted by source vertex. While incoming edges are closely packed in a shard, the outgoing edges of a vertex are dispersed across other shards. In this example, the outgoing edges of $V6$ are dispersed across shard1, shard2, and shard3.

GraphChi relies on this shard organization to process vertices in intervals. It first loads into memory a shard corresponding to one vertex interval, as well as all the outgoing edges of those vertices that may be stored across multiple shards. Updates generated during processing are directly passed to the target vertices through the outgoing edges in other shards in the memory. Once the processing for a vertex interval in a superstep is finished, its corresponding shard and its outgoing edges in other shards are written back to the disk.

GraphChi relies on sequential accesses to disk data and minimizes random accesses. However, in a superstep, only a subset of vertices may become active. Due to shard organization above, even if a single vertex is active within a vertex interval the entire shard must be loaded since the in-edges for that vertex may be dispersed throughout the shard. For instance, if any of the $V3, V4, V5$ or $V6$ is active, the entire shard3 must be loaded. Loading a shard may be avoided only if all the vertices in that vertex interval are not active. However, in real-world graphs, the vertex intervals typically span tens of thousands of vertices, and during each superstep the probability of a single vertex being active in a given

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.



$$
\begin{array}{c|cccccc}
 & V1 & V2 & V3 & V4 & V5 & V6 \\
\hline
V1 & 0 & 4 & 0 & 0 & 0 & 0 \\
V2 & 0 & 0 & 0 & 0 & 0 & 0 \\
V3 & 8 & 4 & 0 & 0 & 0 & 0 \\
V4 & 0 & 0 & 0 & 0 & 0 & 0 \\
V5 & 0 & 0 & 0 & 0 & 0 & 0 \\
V6 & 3 & 5 & 3 & 2 & 1 & 0
\end{array}
$$

	1	2	3	4	5	6	7	8
val	4	8	4	3	5	3	2	1
colIdx	2	1	2	1	2	3	4	5
rowPtr	1	2	2	4	4	4	9	

(a) CSR format representation for the example graph

Shard1: V1

Src	Dst	Val
3	1	8
6	1	3

Shard2: V2

Src	Dst	Val
1	2	4
3	2	4
6	2	5

Shard3: V3-V6

Src	Dst	Val
6	3	3
	4	2
	5	1

(b) GraphChi shard structure for the example graph

Fig. 1: Graph storage formats

interval is very high. As a result, GraphChi in practice ends up loading all the shards in every superstep independent of the number of active vertices in that superstep.

B. Shrinking size of active vertices

To quantify the number of superfluous page loads that must be performed by shard-based graph processing frameworks, we measured the active vertex and the active edge counts in each superstep while running graph coloring application described in section VII over the datasets shown in Table I. For this application, we ran a maximum of 15 supersteps.

Figure 2 shows the active vertices and active edges count as a fraction of the total vertices and edges in the graph, respectively. The x-axis indicates the superstep number, the major y-axis shows the ratio of active vertices divided by total vertices, and the minor y-axis shows the number of active edges (updates sent over an edge) divided by the total number of edges in the graph. The fraction of active vertices and active edges shrink dramatically as supersteps progress. However, at the granularity of a shard, even the few active vertices lead to loading many shards since the active vertices are spread across the shards.

III. CSR format in the era of SSDs

Given the overheads of loading shards from SSDs even with few active vertices in GraphChi, we consider CSR format for graph processing. CSR format has the desirable property that one may load just the active vertex information more efficiently.

CSR format takes the adjacency matrix representation of a graph and compresses it using three vectors. The value vector, *val*, stores all the non-zero values from each column sequentially. The column index vector, *colIdx*, stores the column index of each element in the *val* vector. The row pointer vector, *rowPtr*, stores the starting index of each row (adjacency matrix row) in the *val* vector. CSR format representation for the example graph is shown in Figure 1a. The edge weights on the graph are stored in *val* vector, and the adjacent outgoing vertices are stored in *colIdx* vector. In CSR format, all the outgoing edges connected to a vertex are stored in a contiguous

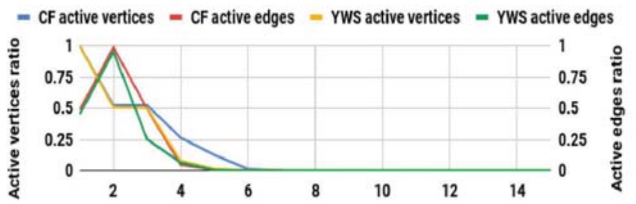


Fig. 2: Active vertices and edges over supersteps, CF and YWS are inputs graphs

location. Accessing only the outgoing edges of the active vertex efficiently is feasible with CSR format. Thus CSR format is suitable for minimizing the number of pages accessed in an SSD when the active vertex set shrinks over time.

A. Challenges for graph processing with a CSR format

While CSR format looks appealing for accessing active vertices, it suffers one challenge. One can either maintain an in-edge list in the colIdx vector or the out-edge list, but not both (due to coherency issues with having the same information in two different vectors). Consider the case that adjacency list stores only in-edges and during the superstep all the updates on the out-edges generate many random accesses to the adjacency lists to extract the out-edge information. Similarly, if the adjacency list stores only out-edges then at the beginning of a superstep each vertex must parse many adjacency lists to extract all the incoming messages.

IV. MultiLogVC: Three Key Insights

In this section we layout three key insights that lead to the design of the MultiLogVC.

A. Avoid random write overhead with logging

To avoid the random access problem with the CSR format, we note that updates to the out-edges do not need to be propagated by accessing the edge list. Instead, these updates can be simply logged separately. The logged message has to store the destination vertex alongside the edge update so that the log can track which destination vertex this message is bound for. Logging the messages has two benefits. First, like any log structure, all the message writes occur sequentially, enabling efficient writes to an SSD for processing later. Second, one can decouple the message logs from the CSR storage format inefficiencies; namely random accesses to either the in-edge or out-edge lists as we described earlier. Thus, we propose to log the updates sent between the vertices, instead of directly updating the edge values in the CSR format. Note that logging has been proposed in prior graph processing works [6], [11]. But as we explain below, prior logging schemes did not exploit the second key observation we make in this work.

Prior works maintain a single log for all the updates in a superstep [11]. But at the start of the next superstep, the entire log must be parsed to find all the messages bound to a given destination vertex. Almost all prior works employ sorting of the log (based on a vertex ID) to efficiently extract all the messages bound for that vertex. In the worst case, the number

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

of updates sent between the vertices may be proportional to the number of edges. Hence, the log itself must be stored in SSD. Even if a small fraction of edges receives an update, the sorting may still overwhelm the host memory, given the size of the graphs. Thus, one has to do external sorting of the message log. Prior works either built a custom accelerator [11] or proposed approaches to reduce the sorting overhead [6].

B. Eliminate sorting overhead with a multi-log

In this work, we exploit the second key observation to get around the sorting constraint. Namely, at any time, only a limited subset of vertices is being processed from the entire graph. Hence, only the incoming messages bound for the currently scheduled vertices must be extracted from the log. Thus, sorting can be narrowed to a few vertices within the log at a time and can be concurrently performed while a previous batch of vertices is being processed.

To enable the above capability, MultiLogVC creates a new multi-log structure. Multiple logs are maintained to hold the messages bound for different vertex intervals. We partition the graph into several vertex intervals and associate one log for each interval. As such, we create a coarse-grain log for an interval of vertices that stores all the updates bound to those vertices. As vertices place outbound messages to destination vertices, the destination vertex interval is used as an index to place the message in that log.

We choose the size of a vertex interval such that typically the entire update log corresponding to that interval can be loaded into the host memory, and used for processing by the vertices in that interval. As the entire update log to a vertex interval can be fitted into the host memory, MultiLogVC avoids the costly step of performing external sort to group the updates bound to a vertex.

C. Reduce read overhead with an edge log

When a vertex interval is processed using a multi-log, one still has to read the outgoing edge information from the CSR so that the processed updates can be properly tagged with the out-edge information for processing in the next superstep. The process of reading the out-edges for each vertex in a vertex interval leads to read amplification, which is the phenomenon where the number of useful bytes is much smaller than the total number of bytes read. Since the minimum read granularity of an SSD is a page (typically 8-16KBs), one has to read the entire page of an edge list vector that stores the outgoing edges of a particular vertex. Given that many real-world graphs exhibit power-law distribution, the vast majority of SSD pages contain the out-edges of multiple vertices. Hence, to read the outgoing edges of a single active vertex, an entire page must be fetched.

We measured the fraction of edge list page data that is actually necessary to process a vertex. The data is shown in Figure 3. As can be seen from this figure, nearly 32% of the accessed pages have greater than 0% and less than 10% of data activity which is necessary for processing. The inefficient use of page data leads to significant read amplification in terms of wasted fetch bandwidth as well as power consumption to move

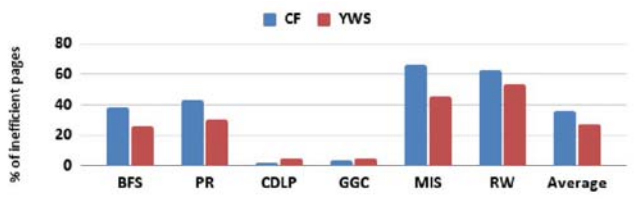


Fig. 3: Accessed graph pages with less than 10% of utilization

Algorithm 1 Overview of a superstep in MultiLogVC

```
1: for all vertex intervals do
2:     $U_{log}$ = LoadLog($V_{in}$)) //Load vertex interval updates
3:     $S_{log}$ = SortNGrpLog($U_{log}$) //Group updates on vertex#
4:     A = ExtractActiveVert($S_{log}$, A)
5:     for each vertex $V_{act}$ in A do
6:         $V_{inf}$=LoadVertexInfo($V_{act}$) // Get $V_{act}$ edges
7:         $V_{actUpdates}$ = ExtractUpdates($S_{log}$, $V_{act}$)//extract
   updates bound for $V_{act}$ from $S_{log}$
8:         ProcessVertex($V_{act}$,$V_{inf}$,$V_{actUpdates}$)
9: function SendUpdate((v, m))
10:    $vint_i$ = vId2IntervalMap(v)
11:    Get $vint_i$'s top page in the log buffer
12:    Append update m to the top page
13:    Flush a page to the SSD on log overflow
```

the data. To tackle this challenge, we design an innovative scheme that places all the outgoing edges of any vertex in a separate edge log. We initiate the out-edge logging process while processing an active vertex in the current superstep. But the challenge is knowing whether the currently processed vertex will be an active vertex in the next superstep. Hence, we log the edges of a vertex that is being processed in the current superstep if that vertex is *likely* to be active again in the next superstep. Thus, the edge log contains the outgoing edges of all likely active vertices to drastically reduce the read amplification overhead.

Note that whether a vertex is active or not in the next superstep is clearly known if there is a message bound for that vertex in the current superstep. Hence, as vertex intervals are processed, the active vertex list for the next superstep becomes increasingly obvious. However, in the early part of a superstep, we need to predict whether a vertex is likely to be active since we do not know whether a vertex processed in a future interval may send a message to the currently processed vertex. We will describe the prediction process in the next section.

V. Multi-log Architecture

In this section, we describe the design and implementation details of the MultiLogVC architecture. There are two components in MultiLogVC: a set of APIs that are used within each superstep to read, write, and sort logs in service of a user-defined graph application. Algorithm 1 shows the pseudo-code of a single superstep actions in MultiLogVC and the various APIs that activate MultiLogVC functionality. The second component is the memory buffers that store various logs that are accessed by the APIs. Figure 4 shows these

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Code snippet of community detection program

```
1: function ProcessVertex(VertexId $V_{act}$, VertexData $V_{inf}$, VertexUpdates $V_{actUpdates}$)
2:     for each update m in $V_{actUpdates}$ do
3:         $V_{inf}$.edge(m.source_id).set_label(m.data)
4:     new_label = frequent_label($V_{inf}$.edges_label)
5:     old_label = $V_{inf}$.get_value()
6:     if old_label ! = new_label then
7:         $V_{inf}$.set_value(new_label)
8:         for each edge in $V_{inf}$.edges() do
9:             m.source_id = $V_{act}$, m.data = new_label
10:            $v_{dest}$ = edge.id(), SendUpdate($v_{dest}$,m)
11:    deactivate($V_{act}$)
```

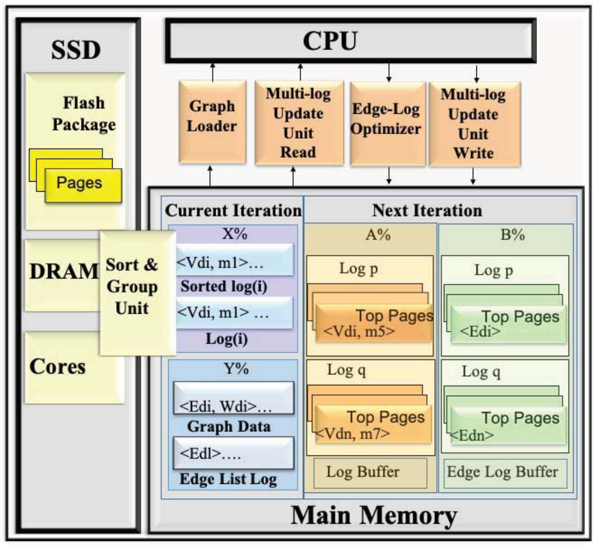


Fig. 4: Layout MultilogVC Framework

modules as highlighted boxes and their interactions with existing OS and SSD functions. An example code snippet for community detection graph application is shown in 2, which uses $ProcessVertex()$ function to provide the vertex-centric computation related to the application.

A. Multi-Log Update Unit

The multi-log update unit is responsible for managing the logs associated with each vertex interval. Its functionality is activated whenever the $ProcessVertex()$ uses $SendUpdate()$ to send message m to a destination v_{dest}. SendUpdate first generates a vertex interval $vint_i$ for the vertex v_{dest} using vId2IntervalMap function. It then appends the message m to the log log_i associated with that vertex interval. Each message appended to the log is of the format $< v_{dest}, m >$. These steps are described in depth below.

1) Generating vertex interval: Typically, in vertex-centric programming updates are sent over the outgoing edges of a vertex, so the number of updates received by a vertex is at most the number of incoming edges for that vertex. Recall that all updates sent to a single vertex interval are sorted and grouped in the next superstep using in-memory sorting process. Hence, the need to fit the updates of each vertex interval in main memory determines the size of the vertex interval. MultiLogVC conservatively assumes that there may be an update on each incoming edge of a vertex to determine the vertex interval size. It statically partitions the vertices into contiguous segments of vertices, such that the sum of the number of incoming updates to the vertices is less than the memory allocated for the sorting and grouping process. This memory size could be limited by the administrators, application programmer, or could be limited by the size of the virtual machines allocated for graph processing. In our implementation, it is limited to a fraction (shown as X% in Figure 4, which is set to 75% as the default value) of the total available memory in our virtual machine (1GB is the default).

2) Fusing vertex intervals: Due to the conservative assumption that there may be a message on each incoming edge, as supersteps progress, the number of messages bound to a vertex interval decrease, and their total size may be less than the allocated memory. To improve memory usage efficiency, MultiLogVC's runtime may dynamically fuse contiguous vertex intervals into a single large interval to process them at once, as we describe later. To enable fusing, MultiLogVC maintains a count of the messages received at each vertex interval to estimate the size of each vertex interval log.

3) SSD-centric log optimizations: For efficient logging, MultiLogVC first caches log writes in main memory buffers, called the multi-log memory buffers. Buffering helps to reduce fine-grained writes, which in turn reduces write amplification to the SSD storage. MultiLogVC maintains memory buffers in chunks of the SSD page size. Since any vertex interval may generate an update to a target vertex that may be present in any other vertex interval, at least one log buffer is allocated for each vertex interval in the entire graph. In our experiments, even with the largest graph size, the number of vertex intervals was in the order of a few ($<$ 5) thousands. Hence, at least several thousands of pages may be allocated in multi-log memory buffer at one time. This log buffer is shown to occupy A% in Figure 4, which is about 5% of the total memory.

For each vertex interval log, a top page is maintained in the buffer. When a new update is sent to the multi-log unit, first the top page of that vertex interval where the update is bound for is identified. As updates are just appended to the log, an update that can fit in the available space on the top page is written into it. If there is not enough space on the top page, then a new page is allocated by the multi-log unit, and that new page becomes the top page for that vertex interval log. MultiLogVC maintains a simple mapping table indexed by the vertex interval to identify the top page.

When the available free space in the multi-log buffer is less than a certain threshold, some log pages are evicted from the main memory to SSD. An evicted log is appended to the corresponding vertex interval log file. Due to the highly concurrent processing of vertices, multiple vertex logs may receive updates, and multiple log page evictions may occur concurrently. To maximize log writeback bandwidth,

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

MultiLogVC spans multiple logs across all available SSD channels. Further, each log is interspersed across multiple channels to maximize the read bandwidth when that log file is read back later. As we make page granular evictions, most of the SSD bandwidth can be utilized. Even when simultaneously writing to multiple logs, they can be parallelized and pipelined due to the interspersing of logs across many channels. Furthermore, we only need to buffer thousands of SSD pages (each corresponding to a log); our host-side buffer size is also small (in 10-100s of MBs). The reduced main memory usage with multiple logs enables us to allocate a significant fraction of the total memory for fetching, sorting, and grouping the updates for processing the superstep, rather than storing the updates for the next superstep.

B. Sort and Group Unit

The sort and group unit is responsible for retrieving the updates from logs and sending the update for processing. Its functionality is automatically activated at the start of each superstep within the MultiLogVC runtime. The first API call is the $LoadLog()$ function, which retrieves the log associated given a vertex interval. The loading process maximizes the SSD read bandwidth since each log is dispersed across all the available flash channels in SSD. After loading a log, if its size is smaller than the memory allocated for sort, the next vertex interval log is automatically fetched. Recall that each vertex interval log maintains a counter which provides a first-order approximation of the log size in that interval. Hence, the loading process uses this approximate size to determine whether there is enough available memory to fetch the next interval log. The process continues until the memory allocated for sorting is full. The fused logs in memory are then sorted based on the v_{dest} field in each log update.

1) Active vertex extraction: Once the sorting process is complete, every vertex V_{dest} that has at least one incoming message becomes an active vertex. Hence, the list of V_{dest} from each log is extracted to update the active vertex set A using $ExtractActiveVert()$ API. Then the next superstep only needs to process the active vertices in A by calling the $ProcessVertex()$ function. To process a vertex, any graph algorithm would need the adjacency list (typically the outgoing edges) of that vertex.

2) Graph Loader Unit: MultiLogVC uses CSR format to store graphs since CSR is more efficient for loading a collection of active vertices. A graph loader unit is responsible for loading the graph data for the vertices present in the active vertex list. Graph data unit maintains the row buffer for loading the row pointer and buffer for each vertex data (adjacency edge lists/weights). The graph data unit loops over the row pointer array for the range of vertices in the active vertex list, each time fetching vertices that can fit in the graph data row pointer buffer. For the vertices active in the row pointer buffer, vertex data required by the application, such as out-edges or in-edges, are fetched from the *colIdx* or *val* vectors stored in the SSD, accessing only the pages in SSD that have active vertex data.

C. Edge-log optimizer

The last component of MultilogVC is the edge-log optimizer module. As we quantified earlier, only a small fraction of the graph pages must be read to fetch an active vertex's outgoing edges. However, due to page granular reads in SSDs, the graph loader unit must read an entire page. When accessing neighbor list for active vertices from the column indices vector, this page granular access requirements lead to read amplification when only a few active vertex neighbors are accessed from a page. All the edge data from inactive vertices that is co-resident in the same page as an active vertex will waste read bandwidth. To reduce this read amplification for such pages, MultiLogVC relies on an edge-log optimizer, which works as follows. At the start of a superstep, the edge log optimizer monitors each vertex v_i that is being processed. Recall that while processing a vertex v_i, all its outgoing edges out_i are fetched. First, the edge log optimizer predicts whether v_i will likely be active in the next superstep. A inactive vertex will be activate in the next superstep if it receives a message on its incoming edge in the current superstep. If a vertex has not received any incoming message yet, the edge log optimizer has to predict the likelihood that v_i will be active in the next superstep; essentially predicting whether v_i will receive messages on the incoming edges at some point during the current superstep.

To predict an active vertex, the edge log optimizer uses the history of active vertices, which is maintained using bit vectors. If the vertex v_i was active at least once in the past N supersteps, it predicts the vertex to be active. More complex prediction schemes were considered, but this simple history-based prediction with N equal to one proved effective.

Once v_i is predicted to be active, the second step is to determine whether there are enough outgoing edges for v_i packed in a page. For vertices with many edges that occupy a substantial percentage of an SSD page, there is no need to extract and log those edges; the page usage efficiency is already quite high. On the other hand, a vertex with low degree (few edges) may store its edges in a page with many other low degree vertices. If all those low-degree vertices that share a page are active in the next superstep, again it is unnecessary to log these edges. But if only a few of these low degree vertices are active, then it is beneficial to log those few edges from active vertices. Since the active vertices are being added dynamically, it is not feasible to accurately determine the page usage efficiency. Hence, the edge log optimizer predicts the page usage efficiency for the next superstep based on the current superstep usage efficiency. Pages that use less than a threshold in the current superstep will be predicted as inefficiently used pages.

Finally, the edge log optimizer logs all the outgoing edges out_i for a predicted active vertex with inefficient page usage into an *edge-log*, indexed with the v_i. The edge-log appends the out-edges of active vertices into a sequential page location. Even if a few predictions are incorrect most of the page data contain out-edges of many active vertices, thereby improving read efficiency. In the next superstep, instead of accessing out-

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

edges from the graph, they can be fetched from the edge-log. In Figure 4, this log buffer is set to occupy B% of the total memory, which is set to 5% as the default value.

Note that unlike message logs, edge logs essentially replicate some of the original graph data. However, by selecting the thresholds appropriately, graph data replication can be limited. In our implementation, we chose a threshold of 10% for determining whether a page is efficiently used. With such a low threshold, the critical observation is that when logging N active vertex outgoing edges into a single edge-log page, one can reduce $N-1$ page reads from the original graph.

D. Supporting the generality of vertex-centric programming

As described in the background section, one of the salient features of MultiLogVC is its ability to support the full spectrum of vertex-centric programming applications. By default, all the messages in the multi-logs are preserved as is, and by using vertex-interval-based multi-logs, we can perform in-memory sorting of each of these logs.

However, some graph algorithms support associative and commutative property on updates. Hence, the updates may be merged into a single update message. MultiLogVC provides an optimization path for such algorithms. For these algorithms, the application developer has to provide the *combine* operator, which may reduce the updates bound to a destination vertex, along with the vertex processing function. The combine operator is applied to all the updates to a target vertex in a superstep before the target vertex's processing function is called. When a combine function is defined, the sort and group unit can optimize the performance automatically by performing the reduction transparently to the user.

E. Graph structural updates:

In vertex-centric programming, graph structure can be updated during the supersteps. Graph structure updates in a superstep can be applied at the end of the superstep. In the CSR format, merging the graph structural updates into the column index or value vectors is a costly operation, as one needs to re-shuffle the entire column vectors. To minimize the costly merging operation, we partition the CSR format graph based on the vertex intervals. Each vertex interval's graph data is stored separately in the CSR format.

Instead of merging each update directly into the vertex interval's graph data, we batch several structural updates for a vertex interval and merge them into the graph data after a certain threshold number of structural updates. As graph structural updates generated during the vertex processing can be targeted to any vertex, we buffer each vertex interval's structural updates in memory. The Graph Loader unit always accesses these buffered updates to fetch the most current graph data for processing accurately.

F. Programming model

We keep the programming model to be consistent with any vertex-centric programming framework. For each vertex, the vertex processing function is provided. The main function logic for a vertex is written in this function. In the current MultiLogVC implementation, the processing function receives the vertex id, including the vertex data (such as vertex value), the list of incoming messages for that vertex, and the vertex adjacency information (in all our applications, we need the outgoing edges). Each vertex processes the incoming messages, sends updates to other vertices, and mutates the graph in some applications. $SendUpdate()$ function implements communication between the vertices, which automatically calls the multi-log update unit transparent to the application developer. The vertex also indicates in the vertex processing function if it wants to be deactivated. If a vertex is deactivated, it will be re-activated automatically if it receives an update from any other vertex. By using multiple logs associated with each vertex interval, MultiLogVC preserves the messages while still enabling in-memory sorting of messages by the sort and group unit. This approach enables MultiLogVC to support the vertex-centric programming model's generality while reducing the sorting and very large log management overheads.

As described earlier, MultiLogVC provides an optional optimization path if the application developer wants to perform a reduction operation on the incoming messages. For the synchronous computation model, updates will be delivered to the target vertex by the start of its vertex processing in the next superstep. In the asynchronous computation model, the latest updates from the source vertices will be delivered to the target vertices, either from the current superstep or the previous one. Vertices can modify the graph structure, but MultiLogVC requires that these graph modifications be finished by the start of the next superstep (which is also a restriction placed on most vertex-centric programming models).

VI. System design and Implementation

We implemented the MultiLogVC system as a graph analytics runtime on an Intel i7-4790 CPU running at 4 GHz and 16 GB DDR3 DRAM. We use 2 TB 860 EVO SSDs [27]. We use Ubuntu 14.04 which runs on Linux Kernel version 3.19.

To simultaneously load pages from several non-contiguous locations in SSD using minimum host side resources, we use asynchronous kernel IO. To match SSD page size and load data efficiently, we perform all the IO accesses in granularities of 16KB, typical SSD page size [8]. Note that the load granularity can be increased easily to keep up with future SSD configurations. The SSD page size may keep growing to accommodate higher capacities and IO speeds; SSD vendors are packing more bits in a cell to increase the density, leading to higher SSD page sizes [7].

We used OpenMP to parallelize the code for running on multi-cores. We use an 8-byte data type for the rowPtr vector and 4 bytes for the vertex id. Locks were sparingly used as necessary to synchronize between the threads. With our implementation, our system can achieve 80% of the peak bandwidth between the storage and host system.

Baseline: We compare our results with the popular out-of-core GraphChi framework. While several recent works have optimized GraphChi [2], [28], [29], [31], they have focused

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

on optimizing algorithms that satisfy the commutative and associative property on their updates. Hence, these approaches tradeoff the generality of GraphChi for improving performance. Since MultiLogVC strives to preserve the generality, we believe GraphChi is the most appropriate comparison framework. On the log-based execution front, we compare our approach with GraFBoost [11], which uses a single log to maintain all updates and exploits the commutative and associative property of graph algorithms to merge the updates to shorten the log. The merging process enables them to perform a relatively efficient out-of-memory sorting. Due to the limitation of GraFBoost, we can only compare MultiLogVC when the algorithms satisfy the constraints of GraFBoost.

While comparing with GraphChi, we use the same host-side memory cache size as the size of the multi-log buffer used in MultiLogVC. Our implementation and GraphChi's implementation limit the memory usage to 1 GB, primarily because the available real-world graph datasets are at most 100 GBs. This approach has been used in many prior works to emulate a realistic memory-graph size ratio [2], [20], [31]. In our implementation, we define memory usage by limiting the total size of the multi-log buffer. GraphChi provides an option to specify the amount of memory budget that it can use. We maximized GraphChi performance by enabling multiple auxiliary threads that GraphChi may launch. As such, GraphChi also achieves peak storage access bandwidth.

Graph dataset: To evaluate the performance of MultilogVC, we selected two real-world datasets, one from the popular SNAP dataset [17], and the other one is a popular web graph from Yahoo Webscope dataset [30]. These are all undirected graphs, and for an edge, each of its end vertices appears in the neighboring list of the other end vertex. Table I shows the number of vertices and edges for these graphs.

Dataset name	Number of vertices	Number of edges
com-friendster (CF)	124,836,180	3,612,134,270
YahooWebScope (YWS)	1,413,511,394	12,869,122,070

TABLE I: Graph dataset

VII. Applications

To illustrate the benefits of our framework, we evaluate two classes of graph applications. The first set of algorithms allows updates to be merged with no impact on correctness, which are proper workloads for GraFBoost. The second set requires all the updates to be individually handled, which can only be evaluated on GraphChi and MultiLogVC.

Merging updates acceptable: BFS, Pagerank (PR) [22]. GraFBoost works well with these algorithms.

Merging updates not possible: Community detection (CDLP) [24], Graph coloring (GC) [9], Maximal independent set (MIS) [26], and Random walk (RW [13]) that are all implemented using the methods in the references.

Additional application details: A vertex in pagerank gets activated if it receives a delta update greater than a certain threshold value (0.4). For random walk, we sampled every 1000^{th} node as a source node and performed a random walk for 10 iterations with a maximum step size of 10.

Due to extremely high computational load, for all the applications, we ran 15 supersteps or less than that if the problem converges before that. Many prior graph analytics systems also evaluate their approach by limiting the superstep count [10], [11], [20], [25]. All the applications mentioned above can be implemented using GraphChi and MulilogVC. However, GraFBoost can only work for associative and commutative applications (Pagerank, BFS).

VIII. Experimental evaluation

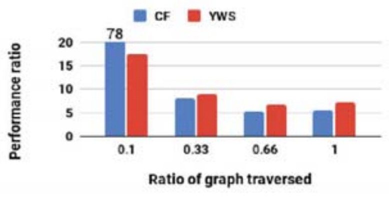
Figure 5a shows the performance comparison of BFS application on our MultiLogVC and GraphChi frameworks. The X-axis shows the selection of a target node that is reachable from a given source by traversing a fraction of the total graph size. Hence, an X-axis of 0.1 means that the selected source-target pair in BFS requires traversing 10% of the total graph before the target is reached. We ran BFS with different traversal demands. The Y-axis indicates speedup, which is the application execution time on GraphChi divided by application execution time on the MultiLogVC framework.

On average, BFS performs $17.8X$ better on MultiLogVC when compared to GraphChi. Performance benefits come from the fact that MultiLogVC accesses only the required graph pages from storage. BFS has a unique access pattern in which as the search starts from the source node; it keeps widening. Consequently, the size of the graph accessed and correspondingly the update log size grows after each superstep. As such, the performance of MultiLogVC is much higher in the initial supersteps and then reduces in later intervals.

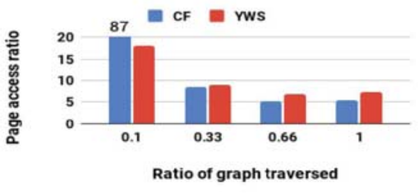
Figure 5b show the ratio of page accesses in GraphChi divided by the page accesses in MultiLogVC. GraphChi loads nearly 90X more data when using 0.1 (10%) traversals. However, as the traversal need increases, GraphChi loads only 6X more pages. As such, the performance improvements seen in BFS are much higher with MultiLogVC when only a small fraction of the graph needs to be traversed. Figure 5c shows the distribution of the total execution time split between storage access time (which is the load time to fetch all the active vertices) and the compute time to process these vertices. The data shows that when there is a smaller fraction of the graph that must be traversed, the storage access time is about 75%; however, as the traversal demands increase, the storage access time reaches nearly 90% even with MultiLogVC. Note that with GraphChi, the storage access time stays nearly constant at over 95% of the total execution time.

Figure 6a shows the performance comparison of pagerank. On average, pagerank performs $1.2X$ better with MultiLogVC. Unlike BFS, pagerank has an opposite traversal pattern. In the early supersteps, many of the vertices are active, and many updates are generated. But during later supersteps, the number of active vertices reduces, and MultiLogVC performs better than GraphChi. Figure 7a shows the performance of MultiLogVC compared to GraphChi over several supersteps. Here X-axis shows the superstep number as a fraction of the total executed supersteps. During the first half of the supersteps, MultiLogVC has similar, or in the case of the

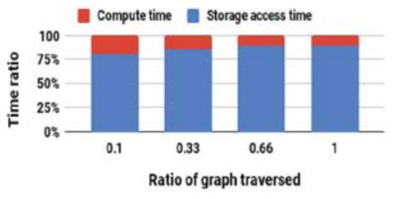
Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.



(a) BFS relative to GraphChi

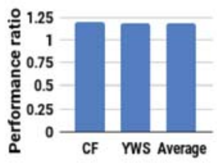


(b) Page access counts ratio relative to MultiLogVC

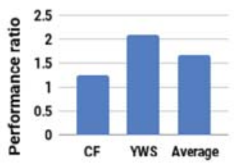


(c) Execution Time Breakdown

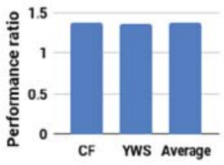
Fig. 5: BFS Application Performance



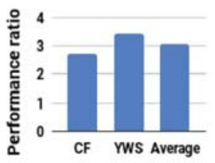
(a) Pagerank



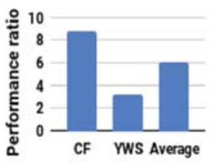
(b) CDLP



(c) GC



(d) MIS



(e) RW

Fig. 6: Application performance relative to GraphChi

YWS dataset, slightly worse performance than GraphChi. The reason is that the size of the log generated is large. But as the supersteps progress and the update size decreases, the performance of MultiLogVC gets better.

Figure 6b shows the performance of community detection, which is similar in behavior to graph coloring (Figure 6c). On average, community detection performs $1.7X$ times better on MultiLogVC over GraphChi. Figure 7b shows performance at supersteps. Like the pagerank application, initially many vertices are active, and in later supersteps fewer vertices are active. In community detection (and graph coloring) application, target vertices receive updates from source vertices over in-edges. Also, active vertices access in-edge weights and store the updates received via source vertices so that a vertex can only send its label if it has been changed. As a result, in the MultiLogVC framework, the community detection application has to access both updates and edge-weights from storage. Whereas in GraphChi, as updates are passed to the target vertices via edge weights, only edge-weights need to be accessed from storage. In later supersteps, MultiLogVC sees few active vertices, and even with the need to access updates and edge-weights separately, it outperforms GraphChi.

Figure 6d shows the performance of maximal independent set algorithm. On average maximal independent set algorithm performs $3.2\times$ better on MultiLogVC when compared to GraphChi. In this algorithm, as vertices are selected with a probability, fewer active vertices are in a superstep. Figure 7d shows the performance over supersteps.

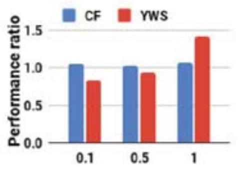
Figure 6e shows the performance comparison of random walk application on the MultiLogVC framework with the GraphChi framework. On average random walk is $6\times$ faster than GraphChi. This algorithm follows the same pattern as BFS. Initially, its active vertices subset is smaller, and gradually it expands and eventually tapers off. Hence, MultiLogVC shows significant improvements in the earlier supersteps, and gradually the performance delta also tapers off.

GraFBoost comparison: Since GraFBoost only works on applications where updates can be merged into a single value, we use pagerank, which satisfies GraFBoost limitation. Figure 8 presents the performance. The Y-axis illustrates MultilogVC performance improvement relative to GraFBoost. To have a fair comparison, we use the same configuration of $1GB$ memory for both of the systems. For GraFBoost, memory usage is limited by CGGROUP instructions. Also, since GraFBoost currently does not support loading only active graph data, the comparison is done based on only the first iteration. Also, as BFS benefits from loading only the active graph data, we do not compare with GraFBoost for BFS. On average, MultilogVC is $2.8\times$ faster than GraFBoost. As can be observed from this figure, Yahoo Web dataset which is a larger dataset shows a significant performance improvement (4 times faster than GraFBoost). The reason is that for larger datasets, the log file size also grows. Therefore, the cost of sorting large logs that do not fit in memory dominates in GraFBoost.

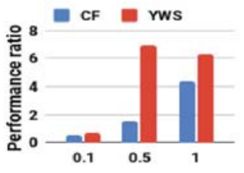
Adapting GraFBoost for applications with non-mergeable updates: We compare the performance of graph coloring application by adapting GraFBoost's single log structure for passing the update messages. As we cannot merge the updates generated to a target vertex into a single value, we need to keep and sort all the updates. For graph coloring, when compared to this adapted GraFBoost, MultiLogVC performs 2.72x and 2.67x for CF and YWS, respectively.

MultiLogVC uses CSR format, which enables accessing only active vertices, but graph updates are costly. Using multiple intervals as separate CSR structures certainly help reduce the update cost of the CSR format. For structural update operations like add edge or vertex, GraphChi and MultiLogVC

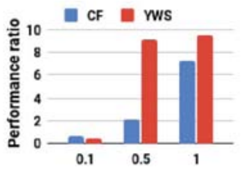
Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.



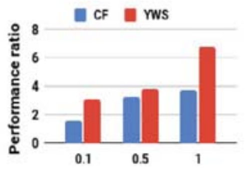
(a) Pagerank



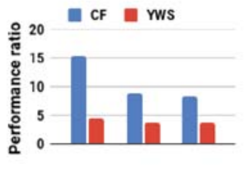
(b) CDLP



(c) GC



(d) MIS



(e) RW

Fig. 7: Application performance comparisons over supersteps

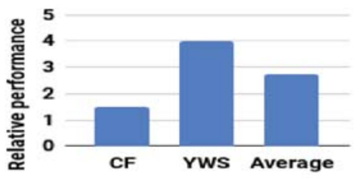


Fig. 8: GraFBoost comparison

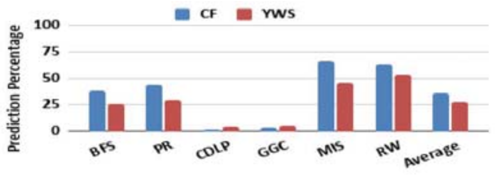


Fig. 9: Predicted inefficient pages

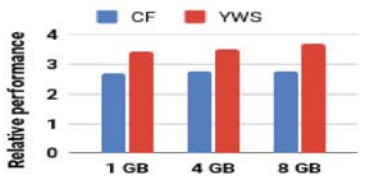


Fig. 10: Memory scalability

tackle the structural updates similarly; namely, they buffer the updates and merge after a threshold.

Edge-log optimizer prediction accuracy: Figure 9 shows the percentage of inefficiently used graph pages (i.e. pages with > 0 and $< 10\%$ of their content utilized) that we predicted correctly. On average, our scheme predicts 34% of the inefficiently used pages. As they converge faster in CDLP and GGC applications, the number of inefficient pages is fewer (shown in Figure 3) concomitantly our history-based prediction model predicts with less accuracy. However, for other applications with higher inefficient pages over several iterations, our prediction accuracy is higher.

Memory scalability: To study the scalability, we conducted experiments with increasing main memory to 4 GB and 8 GB. Figure 10 shows the performance of MultiLogVC over GraphChi for MIS application. As memory size increases, the relative improvement of MultiLogVC over GraphChi stays about the same with about a 5-10% increase in the performance of MultiLogVC when using larger memory.

IX. Related work

Due to the popularity of graph processing, frameworks have been developed in a wide variety of system settings. Examples include Pregel [19] in distributed system setting, Mosaic [18] in single-node setting, and engines such as Galois [5], [21] that supports both single machine and distributed systems.

For vertex-centric programs with associative and commutative functions, where one can use a combine function to accumulate the vertex updates into a vertex value, GraFBoost implements an external memory graph analytics system [11] using a single log. Unlike GraFBoost, our approach maintains multiple logs to minimize the sorting overhead. MultiLogVC supports, complete vertex-centric programming model, which has better expressiveness [3].

X-stream [25] and GridGraph [32] are edge-centric external memory graph analytics systems which aim to sequentially access the graph data stored in secondary storage. However, their efficiency suffers when graphs applications require random and sparse accesses to graph data such as BFS, or programs which require access to adjacency lists for specific vertices such as random-walk.

GraphChi [14] is an external memory based vertex-centric programming system that supports wide range of application models. In this work, we compare with GraphChi as a baseline and show considerable performance improvements. There are several works which extend GraphChi by trying to use all the loaded shard or minimizing the data to load in shard [1], [16], [28]. However, in this work, we avoid loading data in bulky shards at the first instance and access only graph pages for the active vertices in the superstep.

In the context of in-memory graph processing, several works explore the idea of partitioning the graph and using logs to reduce the random DRAM accesses in associative and commutative combine functions based vertex-centric programs [4], [15]. In this work, we design the graph processing framework which can support a complete vertex-centric programming model for flash storage based out-of-core graph processing, designing optimizations that are aware of page-based SSD organization [12] such as edge-log optimization to avoid accessing underutilized SSD pages, page organization based main memory log buffers which can take advantage of available SSD channels to efficiently access the SSD while evicting the log and loading the log.

X. Conclusion

Existing external memory graph processing systems try to minimize the storage access bottleneck by replacing random accesses to storage with sequential requests, but at the cost of loading many inactive vertices in a graph. In this paper, we make a case for using CSR formatted graphs that are more amenable for selectively loading only active vertices in each superstep of graph processing. However, the CSR format leads to random accesses to the graph during the update process. We solve this challenge by using a multi-log update system.

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.

We further develop edge-log optimization to avoid accessing the underutilized graph pages. Over the current state of the art out-of-core graph processing framework, our evaluation results show that MultiLogVC framework improves the performance by up to 17.84×, 1.19×, 1.65×, 1.38×, 3.15×, and 6.00× for the widely used BFS, pagerank, community detection, graph coloring, maximal independent set, and random-walk applications, respectively.

REFERENCES

[1] Ai, Zhiyuan and Zhang, Mingxing and Wu, Yongwei and Qian, Xuehai and Chen, Kang and Zheng, Weimin, Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o, USENIX ATC, 2017.

[2] Ai, Zhiyuan and Zhang, Mingxing and Wu, Yongwei and Qian, Xuehai and Chen, Kang and Zheng, Weimin, Clip: A disk I/O focused parallel out-of-core graph processing system. In IEEE TPDS, 2018.

[3] Apache Flink, Iterative Graph Processing, https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/iterativegraphprocessing.html, 2019.

[4] Beamer, Scott and Asanović, Krste and Patterson, David, Reducing Pagerank Communication via Propagation Blocking, IPDPS, 2017.

[5] Dathathri, Roshan, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali, Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics, In Proc. of PLDI, 2018.

[6] Elyasi, Nima and Choi, Changho and Sivasubramaniam, Anand, Large-scale graph processing on emerging storage devices. In Proc. of FAST, 2019.

[7] NAND Flash Memory, https://www.enterprisestorageforum.com/storage-hardware/nand-flash-memory.html, 2019.

[8] Understanding Flash: Blocks, Pages and Program/Erases, https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/, 2014.

[9] Gonzalez, Joseph E and Low, Yucheng and Gu, Haijie and Bickson, Danny and Guestrin, Carlos., Powergraph: Distributed graph-parallel computation on natural graphs. In Proc. of OSDI, 2012.

[10] Han, Wook-Shin and Lee, Sangyeon and Park, Kyungyeol and Lee, Jeong-Hoon and Kim, Min-Soo and Kim, Jinha and Yu, Hwanjo, TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In Proc. of SIGKDD, 2013.

[11] Jun, Sang-Woo and Wright, Andy and Zhang, Sizhuo and Xu, Shuotao and others, GraFBoost: Using accelerated flash storage for external graph analytics. In Proc. of ISCA, 2018.

[12] Koo, Gunjae, Kiran Kumar Matam, I. Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram, Summarizer: trading communication with computing near storage, In Proc. of MICRO, 2017.

[13] Kyrola, Aapo, Drunkardmob: billions of random walks on just a pc. In Proc. of RecSys, 2013

[14] Kyrola, Aapo and Blelloch, Guy and Guestrin, Carlos, Graphchi: Large-scale graph computation on just a PC, OSDI, 2012.

[15] Lakhotia, Kartik and Kannan, Rajgopal and Pati, Sourav and Prasanna, Viktor, GPOP: A Scalable Cache-and Memory-efficient Framework for Graph Processing over Parts. ACM TOPC, 2020.

[16] Hashemi, Seyedeh Hanieh and Lee, Joo Hwan and KI, Yang Seok, Optimal dynamic shard creation in storage for graph workloads, Google Patents, US Patent App. 16/274,232, 2020.

[17] Leskovec, Jure and Krevl, Andrej., SNAP Datasets: Stanford large network dataset collection, 2014.

[18] Maass, Steffen and Min, Changwoo and Kashyap, Sanidhya and Kang, Woonhak and Kumar, Mohan and Kim, Taesoo, Mosaic: Processing a trillion-edge graph on a single machine. In Proc. of EuroSys, 2017.

[19] Malewicz, Grzegorz and Austern, Matthew H and Bik, Aart JC and Dehnert, James C and Horn, Ilan and Leiser, Naty and Czajkowski, Grzegorz, Pregel: a system for large-scale graph processing. In Proc. of ACM SIGMOD, 2010.

[20] Matam, Kiran Kumar and Koo, Gunjae and Zha, Haipeng and Tseng, Hung-Wei and Annavaram, Murali, GraphSSD: graph semantics aware SSD, In Proc. of ISCA, 2019.

[21] Nguyen, Donald and Lenharth, Andrew and Pingali, Keshav, A lightweight infrastructure for graph analytics, SOSP, 2013.

[22] Pagerank application, https://github.com/GraphChi/graphchi-cpp/blob/master/exampleapps/streamingpagerank.cpp.

[23] Quick, Louise and Wilkinson, Paul and Hardcastle, David, Using pregel-like large scale graph processing frameworks for social network analysis. In Proc. of ASONAM, 2012.

[24] Raghavan, Usha Nandini and Albert, Réka and Kumara, Soundar, Near linear time algorithm to detect community structures in large-scale networks. Physical review, 2007.

[25] Roy, Amitabha and Mihailovic, Ivo and Zwaenepoel, Willy, X-stream: Edge-centric graph processing using streaming partitions. In Proc. of SOSP, 2013.

[26] Salihoglu, Semih and Widom, Jennifer, Optimizing graph algorithms on pregel-like systems, VLDB, 2014.

[27] Samsung SSD 860 EVO 2TB. https://www.amazon.com/Samsung-Inch-Internal-MZ-76E2T0B-AM/dp/B0786QNSBD, 2019.

[28] Vora, Keval and Xu, Guoqing and Gupta, Rajiv, Load the edges you need: A generic I/O optimization for disk-based graph processing. In Proc. of USENIX ATC, 2016.

[29] Xu, Xianghao and Wang, Fang and Jiang, Hong and Cheng, Yongli and Feng, Dan and Zhang, Yongxuan, A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing. IEEE TPDS, 2020.

[30] Webscope, Y. ”Yahoo! altavista web page hyperlink connectivity graph, circa 2002.”

[31] Zhang, Mingxing and Wu, Yongwei and Zhuo, Youwei and Qian, Xuehai and Huan, Chengying and Chen, Kang, Wonderland: A novel abstraction-based out-of-core graph processing system. In Proc. of ACM SIGPLAN Notices, 2018.

[32] Zhu, Xiaowei and Han, Wentao and Chen, Wenguang, GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In Proc. of USENIX ATC, 2015.

Authorized licensed use limited to: Zhejiang Tmall Technology Co.Ltd.. Downloaded on July 02,2022 at 16:07:21 UTC from IEEE Xplore. Restrictions apply.