Large-Scale Graph Processing on Emerging Storage Devices

Nima Elyasi, ***The Pennsylvania State University;*** **Changho Choi,** ***Samsung Semiconductor Inc.;***
Anand Sivasubramaniam, ***The Pennsylvania State University***

https://www.usenix.org/conference/fast19/presentation/elyasi

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

NetApp®

Large-Scale Graph Processing on Emerging Storage Devices

Nima Elyasi[†], Changho Choi[‡], Anand Sivasubramaniam[†]
[†]*The Pennsylvania State University,* [‡]*Samsung Semiconductor Inc.*

Abstract

Graph processing is becoming commonplace in many applications to analyze huge datasets. Much of the prior work in this area has assumed I/O devices with considerable latencies, especially for random accesses, using large amount of DRAM to trade-off additional computation for I/O accesses. However, emerging storage devices, including currently popular SSDs, provide fairly comparable sequential and random accesses, making these prior solutions inefficient. In this paper, we point out this inefficiency, and propose a new graph partitioning and processing framework to leverage these new device capabilities. We show experimentally on an actual platform that our proposal can give 2X better performance than a state-of-the-art solution.

1 Introduction

Graph processing is heavily employed as the fundamental computing platform for analyzing huge datasets in many applications such as social networks, web search, and machine learning. Processing large graphs leads to many random and fine-grained accesses to memory and secondary storage, which is detrimental to application performance. Prior work have attempted to develop optimized frameworks for graph processing either in a distributed system [11, 17, 19, 23, 25] or for a single machine [8, 9, 12, 13, 15, 16, 18, 20, 22, 26], by partially/completely storing the graph data in main memory (DRAM memory).

Recent efforts on single machine approaches aim at storing *Vertex data* in the main memory to serve their fine-grained accesses in the byte-addressable DRAM memory, while the *Edge data* which usually has coarser accesses, is stored in the secondary storage. With growing graph dataset size, even partially storing them on DRAM memory is not cost-effective. On the other hand, emerging storage devices, including currently popular Solid State Drives (SSDs), continue to scale and offer larger capacity with lower access latency, and can be used to accommodate voluminous graph datasets and deliver good performance. However, an SSD's large access granularity (several KB's) is an impediment towards exploiting its substantial bandwidth for graph processing.

Prior works [9, 15] attempt to alleviate this issue by either storing some part of graph data in the main memory or effectively partition the graph data. Such techniques are either designed for the conventional Hard Disk Drives (HDDs) and are not able to saturate an SSD's substantial bandwidth, or are not readily applicable when the vertex data is stored on secondary storage. GraFBoost [13] is a recent fully *external* graph processing framework that stores all graph data on the SSD, and tries to provide global sequentiality for I/O accesses. Despite yielding performance benefits, providing global sequentiality hinders its scalability as graph dataset sizes increase dramatically. On the other hand, since NVMe SSDs deliver comparable performance for random and sequential page-level I/O accesses [2–4], such perfect sequentiality may not be all that essential.

In this paper, we first study the performance issues of external graph processing, and propose a partitioning for vertex data to relax the global sequentiality constraint. More specifically, we address the performance and scalability issues of state-of-the-art external graph processing, where all graph data resides on the SSD. To this end, we devise a partitioning technique for vertex data such that, in each sub-iteration of graph algorithm execution, instead of randomly updating any vertices in the graph, updates occur to only a subset of vertices (which is sufficiently small to fit in main memory).

With our proposed partitioning, after transferring the vertex data associated with a partition into main memory from SSD, the subsequent information required to generate updates –to the vertices present in the memory– will be streamed from SSD. Thus, the fine-grained updates will be only applied to the set of vertices in the memory, eliminating the need for coalescing all the intermediate updates to provide perfect sequentiality. Our proposed enhancements can give more than 2X better performance than a state-of-the-art solution.

2 Background and Related Work

Graph Data Representation: Graphs are represented by (i) *Vertex data* that refers to a set of vertices with vertex attributes including its ID, value, and its neighboring information (i.e., byte offset of its neighbors), and (ii) *Edge data* that contains the set of edges connected to each vertex along with its properties. Edge data is usually stored in a compressed format. A common compressed representation of graph data is called Compressed Sparse Column (CSC) wherein vertex file stores the vertex information along with the byte offset of its neighbors in the edge file.

Programming Model: Due to its unique characteristics, large-scale graph processing is inherently not suited to the parallelism offered by previous parallel programming models. Among different models to facilitate processing of large graphs, *Vertex-Centric* programming model [19] has received much attention, as this iterative model is properly designed to distribute and parallelize large graph analytics. In this model, each vertex runs a *vertex program* which reads its attributes as well as its neighboring vertices, and generates updates to itself and/or its neighbors.

Graph Processing Frameworks: Numerous prior efforts incorporate vertex-centric model and disperse graph data amongst several machines, with each machine storing its portion on DRAM memory, to expedite the fine-grained and random accesses to the graph data. Distributing the graph data, on the other hand, necessitates frequent communications. Such approaches employ various partitioning techniques to minimize the communication overhead [11, 17, 19, 25], and balance the load.

Apart from distributed graph analytic frameworks, single-machine techniques have also been proposed [7–9, 12, 13, 15, 16, 18, 20–22, 26]. When a single machine is used, it may fully or partially store graph data on the secondary storage, and transfer it to main memory in fairly large chunks to achieve high I/O sequentiality. It is common in such techniques to store vertex data on main memory, and edge data on the secondary storage. GraphChi [15], specifically designed for HDDs, splits graph data into different partitions, where partitions are processed consecutively. Their enhancements has two consequences: (i) with increasing graph data size, the number of partitions can proportionally increase, resulting in high I/O costs, and (ii) when only a portion of graph data is required (e.g., when running a sparse graph algorithm), all graph data has to be transferred to main memory. FlashGraph [9] stores vertex data on DRAM memory while edge data resides on an array of SSDs. However, with graph data continuing to grow, even storing the vertex data–which is usually orders of magnitude smaller than edge data– requires considerable amount of expensive DRAM memory. Thus, it is important to consider completely external graph processing approaches.

External Graph Processing: Storing vertex data on SSD has performance and lifetime penalty due to fine-grained I/O accesses. For example, in push-style vertex-centric model, the value of different vertices (e.g., the rank in PageRank algorithm) needs be updated at the end of each iteration. Such updates are usually in the range of a few bytes (e.g., 4 byte integer), whereas the SSD page size is a few kilobytes (e.g., 4KB∼16KB). Apart from its poor performance, an important consequence of the miss-match between the granularity of vertex updates and SSD page size, is its detrimental impact on SSD's endurance.

GraFBoost [13] proposes a sort-reduce scheme to coalesce all the fine-grained updates and submit large and sequential writes to the SSD. In each iteration of GraFBoost after running an *edge program* for the edges connected to a vertex v, a set of updates are generated for the neighbors of v. These updates are in the form of $< key, value >$ pairs, where *key* is the neighbor's ID, and *value* refers to the value of v (source vertex). The number of intermediate updates can be commensurate with the number of edges, denoted as $|E|$, with many duplicate *keys* generated for each destination vertex.

GraFBoost sorts and reduces the $< key, value >$ pairs to convert the fine-grained updates to large sequential SSD writes. Since the number of updates can reach well beyond the size of available DRAM memory, the graph data is streamed from SSD, processed and sorted in main memory in large parts (e.g., 512MB), and then logged on the SSD. Subsequently, these 512MB chunks are streamed from SSD, merge-reduced and written back to the SSD. Despite providing significant benefits, GraFBoost, or any external graph processing approach which tries to provide perfect sequentiality for all vertex updates, incurs high computation overhead. This computation could be avoided for SSDs which provide quite good page-level random access performance, unlike HDDs.

3 Motivation

In this section, we study the performance and scalability issues of GraFBoost, a state-of-the-art external graph processing framework. To investigate its performance, we run various graph algorithms, using different input graphs. For our experiments, we use a system with 48 Intel Xeon cores, 256 GB of DRAM, and two datacenter-scale Samsung NVMe SSDs with 3.2 TB capacity in total, which provide up to 6.4 GB/s sequential Read speed. We run two algorithms, *Breadth First Search* (BFS) and *PageRank* on various input graphs (details can be found in Table 1) including web crawl graph [6], twitter graph [5], and synthetic graphs generated based on Graph 500 benchmark [1]. This synthetic set of input graphs enables us to generate and examine graphs with various size.

Performance Analysis. We give the breakdown of normalized execution time for BFS and PageRank in Figure 1, running on three graphs: `web`, `twitter`, and `kron30`. The latency of different steps of GraFBoost, including (i) reading/writing vertex data, (ii) reading edge data and running edge program, and (ii) the sort-reduce phase, are reported in Figure 1. As

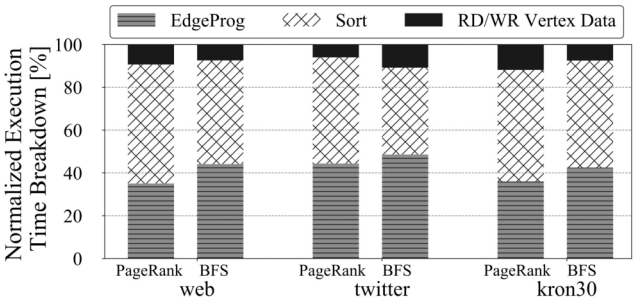

Figure 1: Execution time breakdown of GraFBoost.

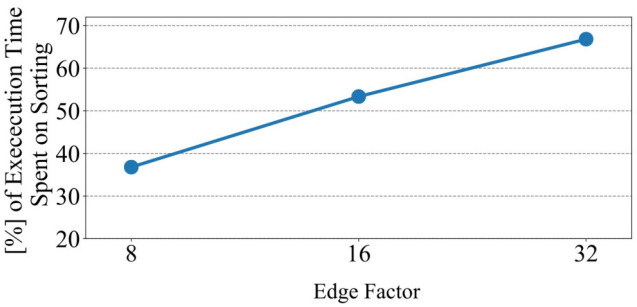

Figure 2: Percentage of execution time spent on sorting.

shown in this figure, the sort phase is the major contributor to the overall execution time, by accounting for nearly 61% of the total execution time for running PageRank on `web` graph. GraFBoost, despite effectively eliminating fine-grained I/O accesses, requires to expend considerable part of its execution time only for the *sorting* phase. In other words, it trades off the additional computation for I/O accesses. This is, in fact, very common in many graph processing frameworks, to minimize the communication/transfer overhead at the expense of adding more computation.

Scalability. To investigate the scalability of GraFBoost, we present a simplified analysis of its execution time. Assuming a graph with N edges, the latency of SSD accesses is linear with respect to N, i.e., $O(N)$. Moreover, Sorting in the memory takes $O(N * log(N))$ to complete, on average. With DRAM access speed k times faster than SSD, if the number of edges grows, such that $log(N) > k$, the sorting phase can dominate the total execution time and hinder its scalability. To quantitatively confirm our analysis, we run PageRank on a synthetic graph with different edge factors (ratio of number of edges to vertices). We report the percentage of time spent in the sort phase, in Figure 2, for `kron` graphs with 1 billion vertices and edge factors of 8∼32. As it is evident, increasing the number of edges results in larger sorting overheads, as more number of updates are generated, which in turn, takes longer time to sort.

Summary. Even though the computation cost that GraFBoost introduces may appear to be an acceptable trade-off for

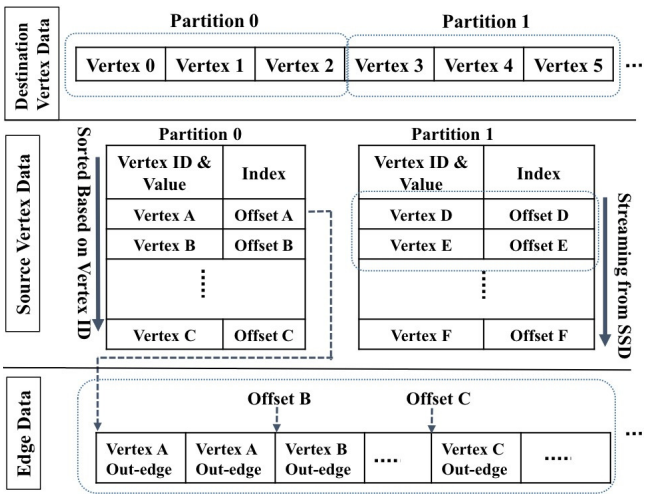

Figure 3: Data structures in our design.

current systems and graph datasets, its benefits are expected to dramatically drop as the graph data sizes grow. Preserving comprehensive sequentiality and sorting of intermediate data, seems to be unnecessary with SSDs providing nearly identical page-level random and sequential access latencies. Instead, if graph vertices can be placed on SSD pages such that each page contains a set of vertices which are likely to be updated at almost the same time, the sorting phase could be eliminated altogether. However, perfectly clustering the graph vertices is known to be an NP-hard problem [14]. In this paper, we aim to provide a local sequentiality which, unlike prior works, does not require any sorting of intermediate updates to achieve lower execution times.

4 Proposed Mechanism

In this section, we describe our proposed partitioning technique that re-organizes vertex data and splits them into several partitions, so that each can fit on a limited DRAM space. The high-level idea is to change the order in which graph vertices are updated, so that at each time, the updates are directed at a subset of vertices residing in main memory. Specifically, we propose to partition the vertex data and process each partition by reading its respective vertex data into main memory, followed by streaming the required edge data from the SSD. Figure 3 shows different data structures employed in our design. Since in each iteration, updates happen to the destination vertices, we (i) split the destination vertices and assign each subset of them to a partition (Destination Vertex Data in this figure), and (ii) store source vertices and their neighboring information –a pointer to the out-edges of each vertex[1]– for each partition, separately (Source Vertex Data in the figure). Lastly, we organize the edge data for each partition as shown in this

[1] e is called an out-edge of vertex u, if $e : u => v$.

figure (Edge Data). Note that, our proposed enhancements are based on the push-style vertex-centric graph processing model.

4.1 Partitioning Vertex Data

There has already been extensive prior work on partitioning graph data. However, they are not well suited for fully external graph processing, due to a number of reasons: (i) some of these studies [9, 15, 18] require all vertex data be present in the main memory when processing the graph, which as prior work [13] shows, they sometimes even fail to finish their execution when the available DRAM space is not enough to store the vertex data; (ii) some others [9, 11, 23, 24, 26] propose 2-D partitioning where graph data is assigned to each partition with the rows/columns corresponding to the source/destination vertices, respectively. These proposals typically do not decouple vertex data from edge data, needing the vertices and edges assigned to a partition be completely present in main memory, or cache, to process it. This constraint results in dramatic rise in the number of partitions which, in turn, accentuates the cross-partition communication overhead. Instead, we devise a mechanism that only requires the vertex data of a partition be present in main memory while edge data can be streamed from SSD, as needed. Based on our proposed greedy partitioning algorithm, destination vertices are uniquely assigned to each partition, whereas source vertices can have duplicates (mirrors) on different partitions. The goal of this greedy partitioning is to minimize the number of mirrors for source vertices while preserving the uniqueness of destination vertices. Based on this partitioning, for each edge $e : u => v$,

- *If* v is already assigned to a partition, u will be added to the same partition, if it does not already exist on that.
- *Else if*, v is not assigned to any partition yet,
  - *If* u is assigned to a set of partitions $\{P_1, P_2, ...\}$, we choose the partition with the least number of edges corresponding to it.
  - *Else*, we assign u and v to the partition with least number of edges corresponding to it.

This partitioning guarantees that each destination vertex is uniquely assigned to a partition and it does not have any mirrors. After this phase, the destination vertex IDs are updated with respect to their new order. These changes are also reflected on the respective source vertices and the edge data. The size of partitions are adjusted such that destination vertices for each partition can fit in main memory. Note that, partitioning is done off-line, as a pre-processing step, latency of which does not impact the execution time.

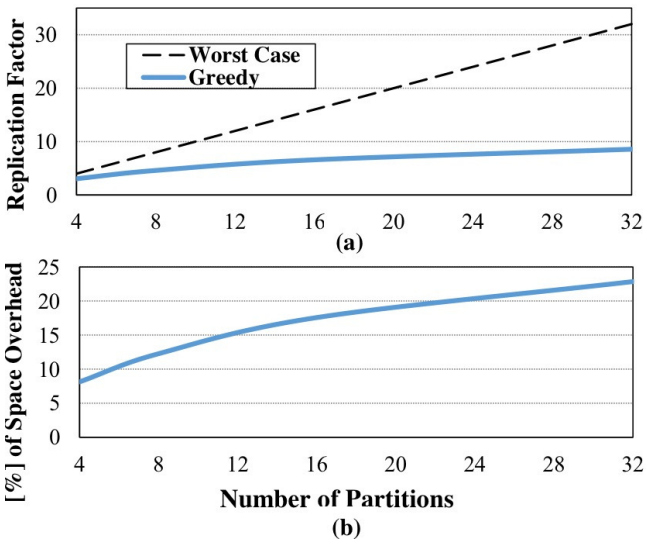

Figure 4: Overhead of the proposed partitioning.

4.1.1 Partitioning Overhead

We study the efficacy of our proposed partitioning, based on the *replication factor*, i.e., the average number of mirrors that each vertex has, and the space overhead. To this end, we run our partitioning algorithm on `twitter` graph, for different number of partitions, and report results in Figure 4. As shown in this figure, with increasing number of partitions, the replication factor increases sub-linearly according to the number of partitions, and it is fairly below the worst case. For instance, with 8 partitions, the replication factor and the space overhead are around 4.5 and 12%, respectively. These overheads happen to be smaller for other graphs listed in Table 1 (3.07 replication factor, on average).

4.2 Execution Model

Different partitions are processed consecutively. For each:
1. The destination vertex data associated with that partition is transferred to main memory from SSD.
2. Source vertex data (their attributes and neighboring information) for this partition, are streamed from SSD in 32 MB chunks. This can be done in parallel with each thread reading different chunks. Decisions regarding which vertex data is currently required to be processed (i.e., is active), can be made either on-the-fly or after the source vertex data is transferred into main memory.
3. After determining the set of active vertices (active vertex list), for each active vertex, byte offset of its neighbors on the edge data file is extracted and the required edges are transferred to main memory. Thus, for a chunk of source data, all the required information to run the graph algorithm exists in main memory, including the source vertex attributes, destination vertices in the current partition, and the neighboring

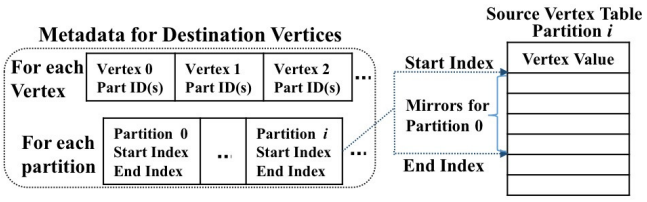

Figure 5: Meta-data for updating mirrors.

```
1.  Read Metadata for Partition P;
2.  For v in P.V:  //P.V: dest. vertex list for P
3.          For part in v.mirror_list_partition:
4.                  out_buf[part].append(v.value);
5.          End_For
6.  End_For
7.  For p_i in {0, Num_Partitions-1}:
8.          Write out_buf[p_i] to SourceTable p_i;
9.  End_For
```

Figure 6: Pseudo code for updating mirrors.

information. This implies that, all the updates generated by this source vertex chunk will happen to the set of destination vertices present in main memory.

4. The graph algorithm runs, and the updates are generated for the destination vertices. As an example, in PageRank, the rank (value) of each source vertex is sent to the destination vertices. The ranks are accumulated in each destination vertex and dampened by a factor specified for this algorithm (e.g., 0.15). In this step, multiple threads are attempting to update elements of the same vertex (destinations) list in memory, which can incur high synchronization cost. Instead, we perform the vertex data updates in two steps: (i) first, threads push updates (in large chunks, e.g., 1MB) to multiple buffers, each dedicated to a portion of the vertex list, and (ii) subsequently, writer threads pull data from these buffers and update their specified portion (similar to Map-Reduce [10] paradigm).

5. When processing for a partition completes, the meta-data (depicted in Figure 5) required for updating its mirrors on other partitions, is read from SSD. The meta-data includes the partition IDs of mirrors of each vertex[2], and the chunk offset for each partition. To minimize the overhead of mirror updates, all source vertex tables store the vertices in the order of their IDs to enables sequential updates to the mirrors.

6. The mirror updates are generated and written on SSD.

[2]We keep Partition IDs in a bitmap to minimize space overhead.

Table 1: Characteristics of the evaluated graph data.

Graph	webgraph	twitter	kron30	kron32
Num Vertices	3.5B	41M	1B	4B
Num Edges	128B	1.47B	17B	32B
Text Size	2.7TB	25GB	351GB	295GB
Rep. Factor	3.7	4.5	1.91	2.2
Space Overhead	10.5%	12%	10.3%	11.5%

4.3 Updating Mirrors

We give pseudo code for mirror updates, in Figure 6. For each vertex in destination vertices, we determine on which partitions its mirrors are located (line 3). In line 4, we insert the value of that vertex to a buffer assigned to destined partition. Lastly, the generated updates are written to the source vertex files on SSD (line 7∼9)[3]. Generating mirrors for different partitions is proportional to the number of destination vertices in each partition, resulting in overall running time of $O(|V|)$, with $|V|$ referring to the number of vertices.

5 Experimental Evaluation

5.1 Evaluation Environment

We evaluate the performance of our proposed mechanism against software version of GraFBoost. GraFBoost also has a hardware implementation, using hardware accelerators. Since the hardware implementation is not available to us, we extract its performance numbers from the original paper [13]. To make a fair comparison, we use the same configuration as GraFBoost: we use 32 processing cores (out of 48 available in our system), 128 GB of memory, and two Samsung NVMe SSDs, totalling 3.2 TB of capacity with nearly identical bandwidth as GraFBoost, i.e., 6.4 GB/s sequential read bandwidth. Similarly, we use the same set of graph data, details of which are reported in Table 1. In this table, we also present the replication factor and space overhead of our partitioning technique, for 8 partitions[4], as it is sufficient for the evaluated graph datasets.

5.2 Evaluation Results

Figure 7 shows the amount of reduction in total execution time (higher is better) for PageRank and BFS, for our proposal (**V-Part**), and software and hardware versions of GraFBoost (**GraFSoft** and **GraFHard**). All performance numbers are normalized to that of GraFSoft. We also show the execution time (in seconds) for PageRank and BFS algorithms, for **GraFSoft** and **V-Part** in Figure 8. As illustrated in these two figures, our proposed partitioning provides better performance

[3]This can be done in parallel for mirror updates on different partitions.

[4]We fix the memory size assigned to a partition's vertex data (e.g., 2GB), and find the proper number of partitions, accordingly.

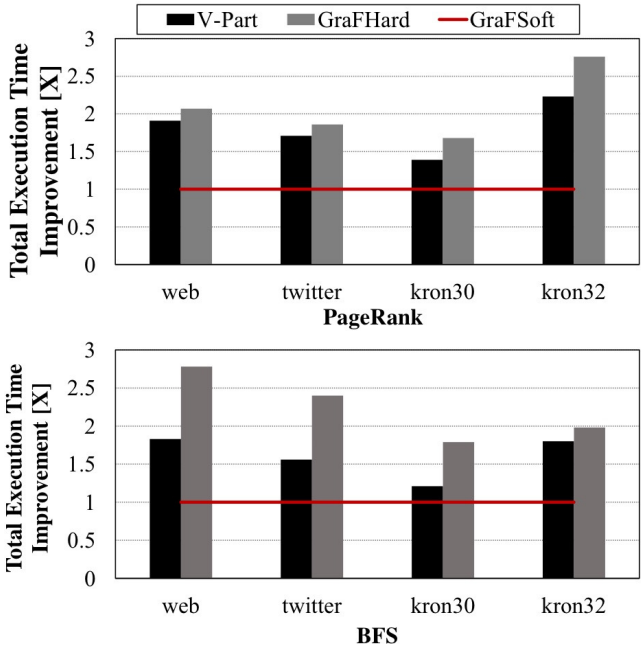

Figure 7: Execution time improvement results.

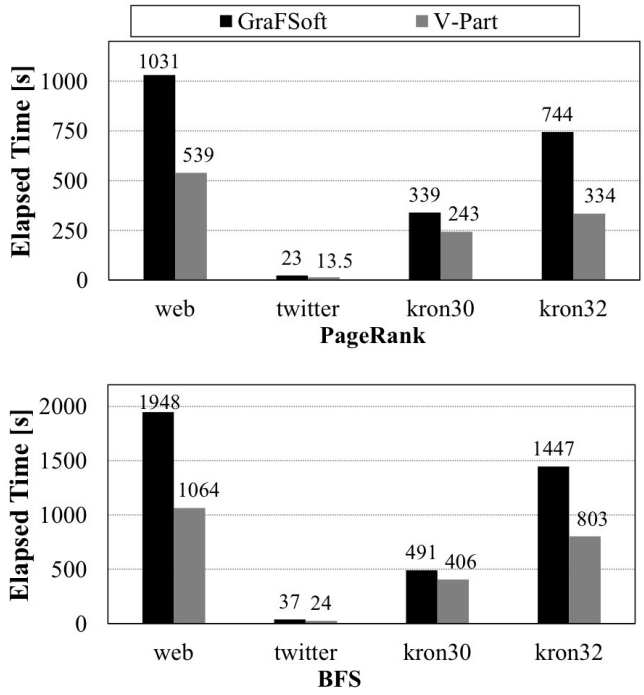

Figure 8: Execution time for (i) a PageRank iteration, and (ii) BFS.

than GraFSoft by around 2.2X (when running PageRank on `kron32`), and 1.8X and 1.6X, on average, as a result of eliminating the burdensome sorting phase of GraFBoost when running PageRank and BFS algorithms, respectively. Moreover, our proposed approach reaps higher benefits when the graph

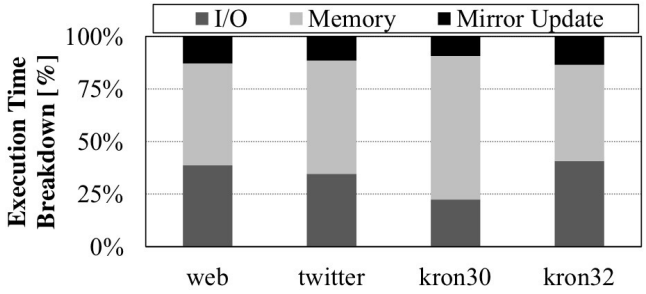

Figure 9: Execution time breakdown for PageRank.

size is larger (`web` and `kron32`). As shown in Figure 7, our optimizations can get very close to GraFHard performance in some cases (e.g., for PageRank on `web`), without incurring any of its hardware and implementation costs. In fact, our mechanism can also use the hardware accelerators to off-load some computation to provide even higher benefits, which we leave it to future work.

In Figure 9 we present the breakdown of execution time for PageRank algorithm. This figure reveals the contribution of each part to the total execution time, including SSD accesses (**I/O**), processing the in-memory graph data (**Memory**), and the time spent on updating mirrors (**Mirror Update**). The **I/O** part does not include SSD accesses for updating mirrors (this part is calculated in Mirror Update). As shown in this figure, the extra work that is introduced to the system for updating mirrors, is less than 15% across the evaluated graphs (even less than 10% in some cases such as `kron30`). This figure also demonstrates that, despite common wisdom, I/O is not the main contributor to the total execution time in graph processing. In some cases, memory accesses delays the processing time more than I/O. Incorporating more efficient caching and pre-fetching techniques, can help lower the memory overhead.

6 Conclusion

In this paper, we study the performance and scalability issues of external graph processing, and devise a mechanism to partition graph vertices to alleviate extra computation overhead of state-of-the-art external graph processing. Our optimizations yield significant performance benefits compared to the state-of-the-art, with more than 2X reduction in total execution time.

Acknowledgements

We thank Tim Harris, our shepherd, and the anonymous reviewers for their constructive feedback. This work has been funded in part by NSF grants 1763681, 1714389, 1629915, 1526750, 1439021, 1302557, and a DARPA/SRC JUMP grant.

References

[1] Graph500 benchmarks. https://graph500.org/.

[2] Intel nvme ssd. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4511-series.html.

[3] Micron nvme ssd. https://www.micron.com/products/solid-state-storage/bus-interfaces/nvme-ssds#/.

[4] Samsung enterprise nvme ssd. http://www.samsung.com/semiconductor/products/flash-storage/enterprise-ssd/.

[5] Twitter graph dataset. http://law.di.unimi.it/webdata/twitter-2010/.

[6] Web data commons. http://webdatacommons.org/hyperlinkgraph/.

[7] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145, Sep. 2018.

[8] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. Venus: Vertex-centric streamlined graph computation on a single pc. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1131–1142. IEEE, 2015.

[9] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58, 2015.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[12] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[13] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018.

[14] David G. Kirkpatrick and Pavol Hell. On the completeness of a generalized matching problem. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 240–245, New York, NY, USA, 1978. ACM.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[16] Hang Liu and H Howie Huang. Graphene: Fine-grained io management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300. USENIX Association.

[17] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, United States, 2010. AUAI Press.

[18] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[20] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*,

SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[21] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[22] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *USENIX Annual Technical Conference*, pages 507–522, 2016.

[23] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[24] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 544–557. IEEE, 2018.

[25] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[26] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.