An Edge-Set Based Large Scale Graph Processing System

Li Zhou
The Ohio State University
Columbus, OH 43210, USA
zholi@cse.ohio-state.edu

Yinglong Xia, Hui Zang
Huawei Research America
Santa Clara, CA 95050, USA
{*yinglong.xia, hui.zang*}*@huawei.com*

Jian Xu, Mingzhen Xia
Huawei Technologies, Ltd.
Shenzhen, Guangdong, China
{*xujian, xiamingzhen*}*@huawei.com*

***Abstract*—Next generation analytics will be all about graphs, though performance has been a fundamental challenge for large scale graph processing. In this paper, we present an industrial graph processing engine for exploring various large scale linked data, which exhibits superior performance due to the several innovations. This engine organizes a graph as a set of edge-sets, compatible with the traditional edge-centric sharding for graphs, but becomes more amenable for large scale processing. Each time only a portion of the sets are needed for computation and the data access patterns can be highly predictable for prefetch for many graph computing algorithms. Due to the sparsity of large scale graph structure, this engine differentiates logical edge-sets from the edge-sets physically stored on the disk, where multiple logical edge-sets can be organized into a same physical edge-set to increase the data locality. Besides, in contrast to existing solution, the data structures utilized for the physical edge-sets can vary from one to another. Such heterogeneous edge-set representation explores the best graph processing performance according to local data access patterns. We conduct experiments on a representative set of property graphs on multiple platforms, where the proposed system outperform the baseline systems consistently.**

***Keywords*-graph; parallel; prefetch; edge-set**

I. Introduction

It is well known that a great variety of big data applications are naturally modeled as large scale graph analytics [1] [2]. It is partially because the entities within many big data applications are typically connected with each other, where the connections form the edges of a graph of the entities. Besides, the intuitiveness of graph-based modeling is amenable to analysis algorithm design. This explains that graph has been a critical component in existing big data computing frameworks, such as Giraph in Hadoop, GraphX in Spark, and Gelly in Flink, etc. [3] [4]

However, compared to many other big data subsystems, the graph processing system imposes significant performance challenges that adversely impact the adoption of the useful technology in some big data scenarios. For example, one of the challenges is poor data locality due to irregular data access. Therefore, graph processing is not bounded by the computational capability of a platform, but the IO latency [5] [6]. This motivates some single machine solution for relatively large scale graph processing, where both the disk and memory resources are leveraged for processing, such as GraphChi [7] and XStream [8]. Our system leverages some idea in this field but achieves much better performance. Such work paves an approach for large scale graph computing; however, those solution still faces challenges that traversal along the graph structure, such as the breadth-first search (BFS).

Our contributions in this paper consist of: (1) a novel edge-set based representation for large scale graph processing that is naturally related to the parallel sliding window (PSW) and is straightforward to be handled when exchanging/prefetching data between memory and disk; (2) a method to store multiple sparse edge-sets with affinity into the same physical block, so that the data locality can be improved; (3) an approach to incrementally update graph structure while keeping data consistent across edge-sets; and (4) a multi-modal representation of an edge-set to incorporate with various graph processing algorithms.

II. Edge-set based Graph Processing System

A. Overview

The architecture of the proposed graph processing system is shown in Figure 1. Basically, the edge-set generator converts graph data into a set of edge-sets, each consisting of a group of edges. Graph analysis algorithms are implemented using the same programming model as that in GraphChi and the scheduler will load/preload corresponding edge-sets for processing. If modified in the edge-set modifier, the resulting edge-set will be persisted onto the storage by the evictor. The in-memory edge-set manager maintains the edge-sets that are currently cached in the edge-set buffer and decides which to evict according to an alternated LRU policy that considers the edge-set to prefetch. The edge-set buffer hosts the edge-sets under processing and those prefetched.

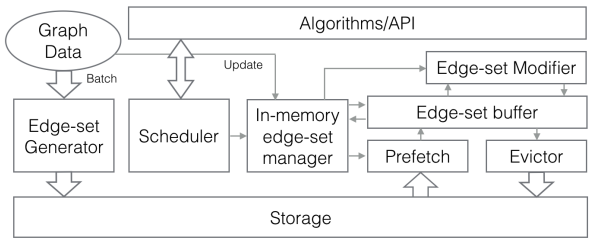

Figure 1. System architecture

B. Edge-set Representation

The edge-sets are naturally related to the parallel sliding window (PSW) in [7], but more flexible. For example, in Figure 2, an input graph is represented as three edge lists known as *shards*, each consisting of all the edges with the destination vertex in a certain range. We show the ranges on top of the shards. To traverse a graph, the PSW works in an iterative manner. The yellow zone covers the data to be processed in the current iteration. It is worth noting that, at the i-th step the yellow zone exactly corresponds to the i-th row plus the i-th column of the blocked adjacency matrix. Therefore, to traverse a graph, it is equivalent to scan the blocked adjacency matrix top-down and left-right simultaneously. Such regularity implies an approach to efficiently prefetch data. Note that the graph sharding in [7] corresponds to the vertical-only partitioning of the adjacent matrix, which is incapable to address celebrity vertices with extreme dense incoming edges, but such an issue does not exist for our edge-set based approach where the matrix is partitioned both vertically and horizontally.

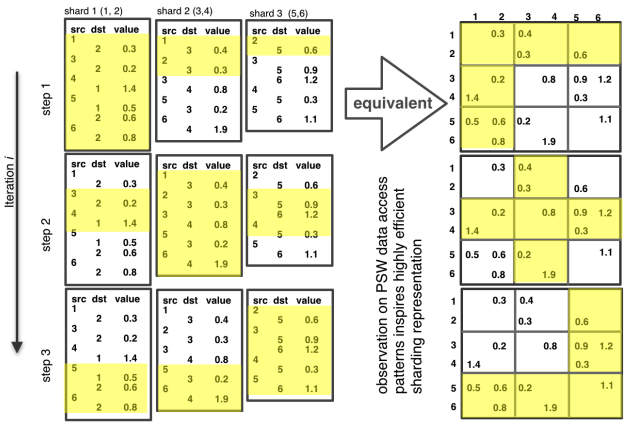

Figure 2. PSW in terms of edge-sets

Since all graph algorithms can be implemented using PSW under the gather-apply-scatter (GAS) model (or its variants) [7] and PSW is nothing but all the edge-sets on the same column in the blocked adjacency matrix, we conclude that the edge-set representation of a graph is generic. To generate the edge-sets, it is even more straightforward than that in GraphChi where a global sorting is required. In our case, we scan the edge list once to determine the vertex degrees and then we divide the vertices into a set of range by evenly distributing the degrees. Then, we scan the edge list again and allocate each edge to an edge-set according to the ranges where source and destination vertices fall into. Note that both scans can be conduct in divide-and-conquer manner. Thus, given p parallel threads, the complexity under PRAM is given by $O(m/p)$, where m is the number of input edges. In contrast, GraphChi sorts all edges and then generates the shards. Given sufficient memory (i.e. a single shard for GraphChi) the complexity is $O(m \log m) > O(m/p)$. Note that GraphChi actually utilizes the radix sort with complexity $O(km)$, but theoretically $k \leq \log m$. In practice, we also observed improved parallelism and performance for our proposed approach.

C. Consolidation

The edge-set generator shown in Figure 1 can merge small edge-sets. The sparsity nature of real large scale graph can result in some tiny edge-sets that consist of a few edges each, if not empty. Loading or persisting many such small edge-sets is inefficient due to the IO latency. Therefore, it makes sense to consolidate small edge-sets likely to be processed together, so that we can potentially increase the data locality. Consolidation can occur between edge-set next to each other horizontally, vertically, or both. We consolidate edge-sets using the following heuristic method. For the sake of simplicity, we look at the horizontal consolidation only. First, we determine a bound B for the merged set as follows: let k denote the page size of the platform, in term of the number of bytes, and s the size of an edge, then the bound is given by $\lceil \frac{k}{s} \rceil$, which ensures that the resulting set is aligned with the system page, leading to improved IO efficiency. Second, for each edge-set $s_{i,j}$ smaller than the bound, i.e. $|s_{i,j}| < B$, where i, j are the indices of the edge-set in the corresponding adjacency matrix with $N * N$ blocks, it identifies its horizontal neighbor that minimizes the size of the resulting set if merged:

$$\tilde{s} = \min_{j' \in \{j-1, j+1\}, 0<j<N} (|s_{i,j}| + |s_{i,j'}|)$$

If $\tilde{s} < B$, it proposes to merge with the selected neighbor. If two edge-sets select each other, then they are merged. The neighbors of the merged set are the union of the neighbors of the two. We continue the consolidation process repeatedly until no merge occurs anymore. In Figure 3, we merge the neighbor sets as long as the size of the merged set is no more than the given bound, say 4 edges. As a result, edge-sets 1, 2, and 3 are consolidated. Similarly, edge-sets 4 and 5 are merged, and also 8 and 9.

The horizontal consolidation improves data locality especially when we visit the out-going edges of vertices. We can also merge the edge-sets vertically, which benefits the information gathering from the parents of a vertex. Note that if all the edge-sets in a row (column) are merged, it is equivalent to have the out-going (in-coming) edge list of the vertex. Note that the edge-set consolidation is transparent to users, that is, the users will still see 9 edge-sets when implementing graph algorithms; but physically there are only 5 edge-sets physically stored on disk. The proposed system maintains the mapping between the logical edge-sets and the physical edge-sets. Once a logical edge-set is prefetched, the system is aware that all logical edge-sets co-existing in the

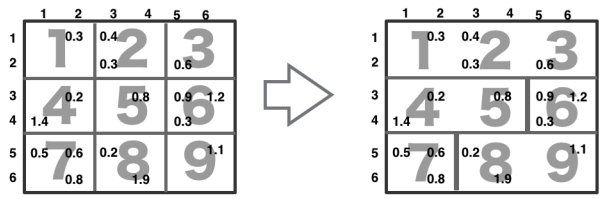

Figure 3. Horizontal consolidation of logical edge-set to improve data locality

same physical edge-set become available in memory, which are likely being processed immediately. Thus, the temporal data locality is improved.

D. Multi-modal Organization

We allow multi-modal data organization for the edge-sets, because of the impact of organization formats on particular graph computing algorithms. We take two formats as an example: The coordinate format (a.k.a. COO) in our context is simply a list of edges, each having a source vertex ID, a destination vertex ID, and some attribute on the edge; while the compressed sparse row (CSR) in our context sorts COO according to the source vertices and then compresses the list by eliminating the repeated source vertices. Both can be found in literature of sparse matrices and graphs. The impact of COO and CSR on performance varies according to the graph processing algorithms. Specifically, for the same input graph, we observed better performance for performing PageRank using COO than CSR, although CSR helps the IO a little bit due to the compression. However, for performing breadth-first search (BFS), CSR shows higher noticeable advantage. The reason is that the CSR allows us to locate a vertex quickly as it is sorted; while for COO we have to filter the edge-set when seeking a particular vertex. Although due to high sparsity, COO may help save the memory required to present a graph than CSR where each vertex has a pointer. Note that in PageRank we visit all the edges in each iteration of the algorithm, regardless the order of the edges; while in BFS, we must follow the graph topology to visit the neighbors of the vertices visited in the last iteration.

E. Scheduling and Prefetching

The scheduler shown in Figure 1 applies the user-defined vertex program to the graph and coordinates with the *in-memory edge-set manager*. The manager maintains buffer of edge-sets. The scheduler notifies the manager which edge-sets will be processed, according to the data access pattern discussed in Section II-B, and the edge-set manager informs the prefetch component to load those edge-sets, as long as the buffer is not full. In the meanwhile, the *evictor* dumps the edge-set that are least recently used. The *edge-set modifier* update edges and/or its property. Note that the scheduler is aware of the spatial/temporal data locality. If an edge-set is already loaded, it won't be load again.

Name	Vertices	Edges
Kronecker (22)	4.1M	34.1M
Kronecker (24)	16.7M	165.2M
Kronecker (26)	67.1M	799.8M
LDBC-1000K	1M	28.8M
Twitter-2010	41.7M	1.4B

Table I
DATASETSS DESCRIPTION

III. EXPERIMENT

We preliminarily evaluated our system on two platforms: One is a Mac Mini with a dual-core 3 GHz Intel i7 processor, 16 GB memory, and 1 TB HDD. The other is a Dell Precision Tower with 8-core 3.4 GHz Intel i7 processor, 64 GB memory, and 1 TB HDD. The OSs are Mac OS X and Ubuntu 14.04 LTS, respectively. Both synthetic and real datasets were utilized (see Table I). The three Kronecker graphs were generated with the seed matrix $[0.9, 0.6; 0.6, 0.1]$ and numbers of iteration are 22, 24, and 26, respectively. One of the workloads was the PSW-based graph traversal for 10 times. Note that such workload is equivalent to perform PageRank on the graph for 10 iterations [7]. We took GraphChi [7] as a baseline and implemented the above workload in both the baseline system and our proposed system. In addition to PageRank, we also implemented BFS and single-source-shortest-path (SSSP) in GraphChi and compared its performance with that of our system. Note that BFS and SSSP have very different characteristics compared to PageRank in terms of graph data access behaviors.

A. Evaluated Workloads

We observed highly promising performance improvement against our baseline methods in our preliminary experiments. In Figure 4, we illustrate the execution time of our workload against GraphChi. The efficiency improvement was quite significant, approximately 3.6x $\sim$ 10.4x faster. We achieved such speedups because 1) our system eliminates vertex-centric graph reconstruction in GraphChi that results in significant memory allocation and release repeatedly; 2) our system explores data parallelism in an edge-set by processing multiple vertices simultaneously.

Our system achieved $10\% \sim 30\%$ performance improvement over GraphChi on the preprocessing phase, primarily because our system requires no global sorting. To combine the preprocessing and the workload of 10 PSW-based traversal, we achieve 1.5x $\sim$ 3.4x overall speedup as shown in Figure 5.

We also implemented PSW-based BFS and SSSP in GraphChi and compared with our system. In Figure 6, we illustrate the execution time of PSW-basded BFS against GraphChi. The root nodes were randomly chosen, and average execution time was used to compare the performance.

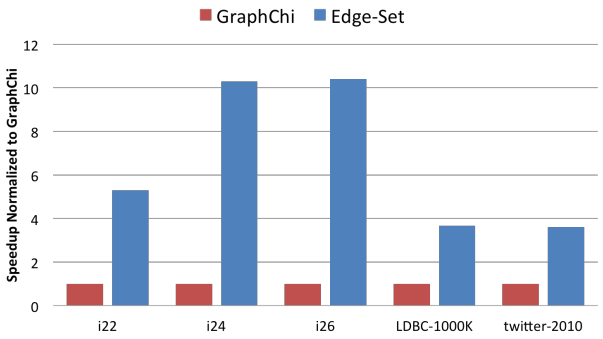

Figure 4. Performance improvement of PageRank against GraphChi.

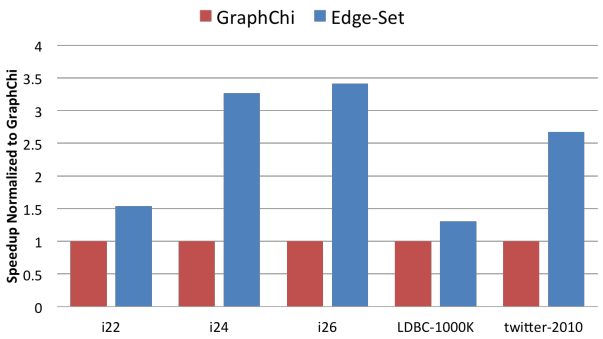

Figure 5. Performance improvement of PageRank against GraphChi including preprocessing.

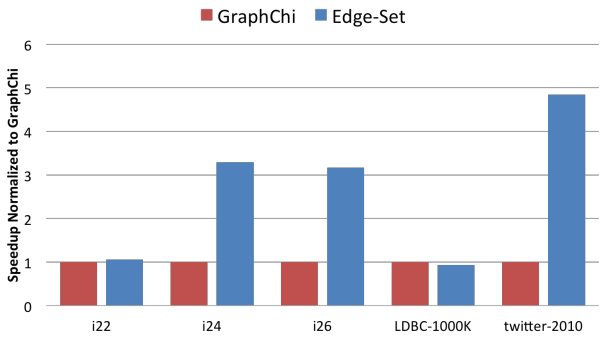

Figure 6. Performance improvement of BFS against GraphChi.

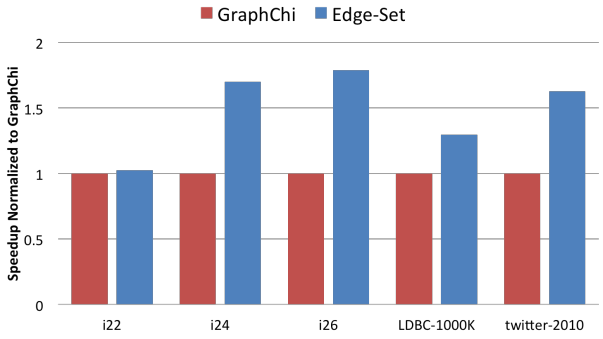

Figure 7. Performance improvement of BFS against GraphChi including preprocessing.

When the input dataset is small enough to fit into the memory, GraphChi shows similar performance with our system. This is because the IO time dominated the execution phase while the graph reconstruction time was relatively small. In addition, it is natural to traverse the graph under vertex-centric model after reconstructing the whole graph in memory. Our system shows potentials with large graphs, it presented up to 5x faster execution with increasing input dataset size. Such speedups of our system are mainly from eliminating graph reconstruction overhead in GraphChi. The data parallelism was also improved by processing multiple verticse simultaneously when accessing each edge-set. To combine the preprocessing and the workload of PSW-based BFS, we achieve 1x $\sim$ 1.8x overall speedup as shown in Figure 7.

In Figure 8, we illustrate the execution time of PSW-based SSSP against GraphChi. We implemented Bellman-Ford algorithm which computes the the shortest paths from a single source vertex to all of the other vertices in a weigeted graph. It traverse all the edges |V|-1 times where |V| is the number of vertices in the graph, or ends when no vertices distance are changed since last iteration. The source vertices were randomly chosen, and the average execution time were calculated and compared with GraphChi. The efficiency improvement was quite significant, approximately 3.7x $\sim$ 16.2x faster over the baseline. The speedups vary since the iterations required to end may vary with different source vertices. To combine the preprocessing and the workload of PSW-based SSSP, we achieve 1.5x $\sim$ 2.9x overall speedup as shown in Figure 9.

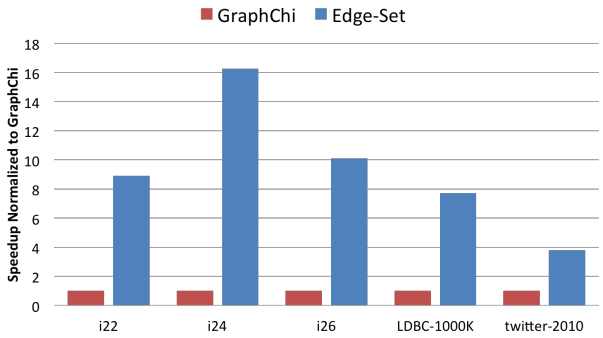

Figure 8. Performance improvement of SSSP against GraphChi.

In Table II, we illustrate the total time breakdown for the workloads. Our system outperform GraphChi for all the preprocessing, load and reconstruction (which is not required in our system), and computation phases. Note that the experiments for BFS and SSSP randomly chose source vertices and presented the average time. Both GraphChi and

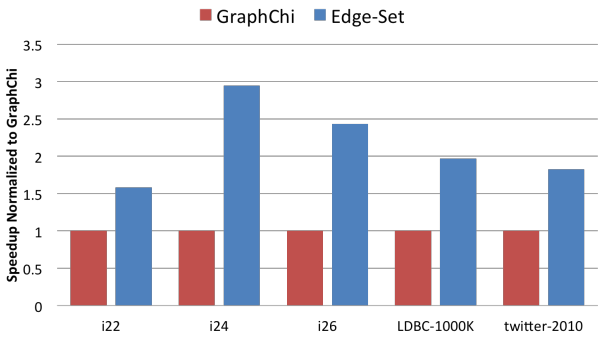

Figure 9. Performance improvement of SSSP against GraphChi including preprocessing.

	App.	Prep.	Load+Reconstr.	Comp.
GraphChi	PageRank	1184.64	2535.76	1030.76
	BFS	958.991	197.6692	82.5688
	SSSP	966.342	217.1342	82.9308
Edge-Set	PageRank	786.199	431.693	557.81
	BFS	703.41	46.1713	11.7145
	SSSP	712.216	41.5668	25.1802

Table II
TOTAL TIME BREAKDOWN RUNNING TWITTER-2010 BY GRAPHCHI AND EDGE-SET.

our system require preprocessing to generate shards or edge-sets, which is a one-time task given a constant graph.

B. Scalability and Performance

We conducted preliminary experiments to evaluate the impact of edge-set prefetch (Section II-E), consolidation (Section II-C), and the edge-set organization (Section II-D). We noticed that a larger buffer size usually results in higher performance improvement. The prefetch contributed up to 6% of the overall execution time improvement in our observation. We evaluated the edge-set consolidation with various upper bounds of the physical edge-set size using the three Kronecker graphs. The results are shown in Table III. Figure 10 presents the comparison of the execution time (including preprocessing) on Kronecker (26) between two graph representative formats COO and CSR. COO showed improved performance for the PSW-based workload since PSW visits all (active) edges (see Figure 10). Note that CSR shows advantages for BFS-like traversals since it is essentially an indexed COO (with compression) that helps in locating a specific vertex, however it may require more memory than COO in a highly sparse graph which degrades the IO efficiency and load balance issue also should be considered when using CSR. In overall, COO format obtains 8.4x speedup on execution time over CSR and 2.9x if the preprocessing is considered.

In Figure 11, we illustrate the scalability of our system. The execution time was normalized to 2 cpus GraphChi,

Dataset	Performance Improvement
Kronecker (22)	4.8% $\sim$ 9.3 %
Kronecker (24)	3.2% $\sim$ 9.2 %
Kronecker (26)	2.8% $\sim$ 8.6 %

Table III
IMPACT OF EDGE-SET CONSOLIDATION

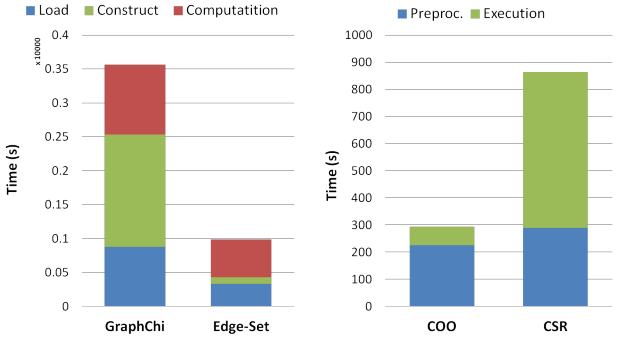

Figure 10. Left: Execution time breakdown on Pagerank running twitter-2010. Right: Impact on edge-set multi-modal organization on performance (COO vs. CSR).

and presented the scalability of both load/reconstruction and computation time. GraphChis performance is limited by the IO bandwidth and graph reconstruction. Increasing the number of cores can improve the graph load and reconstruction, but benefit for computation gained from parallelism is small. Our system utilizes the data parallelism by processing multiple vertices simultaneously in a edge-set. It shows better scalability on computation time with increasing number of cores. This experiment started with two-core since our system was optimized for multicore system, and was compared up to four-core to present a fair comparison since GraphChi targeted on four-core Mac PC. We noticed that for application like BFS, GraphChi showed similar or better computation time with small size graphs. It is because tremendous time was spent to reconstruct the graph into vertex-centric model. However our system still presented better overall performance by eliminating the graph reconstruction and utilizing data parallelism in a edge-set. Our 2-cpu system achieved 2.0x $\sim$ 3.3x overall speedup over 4-cpu GraphChi.

Our system exhibited higher IO efficiency than the baseline. In Figure 12, we presented the disk read bandwidth regarding twitter-2010 over time. Our system showed improved bandwidth usage clearly, up to 2x aggregate bandwidth. GraphChi has disruptive IO efficiency due to the long latency of graph re-construction. This observation explained the high speedup of our system in the experiments.

IV. CONCLUSION AND FUTURE WORK

We presented our undergoing project on a highly efficient graph processing engine. The preliminary experiments show

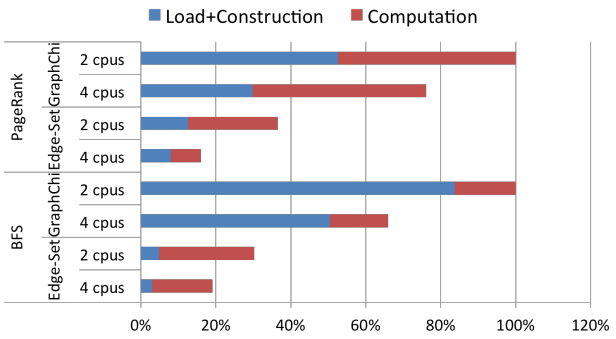

Figure 11. Relative runtime when varying the number of threads used by GraphChi and Edge-Set. Experiment was done on a Linux machines with four cores.

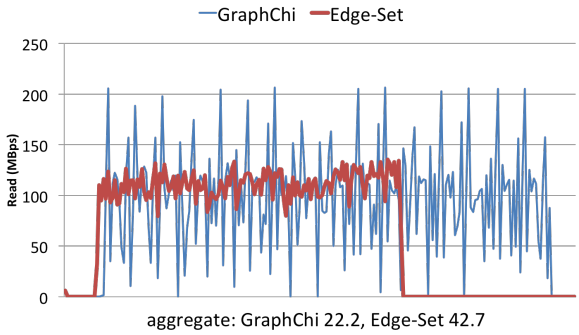

Figure 12. Disk read bandwidth over total run time by GraphChi and Edge-Set. Edge-Set showed up to 2x aggregate bandwidth and more constant IO usage.

highly promising results, such as over 10x speedup on traversing a graph using parallel sliding window. As an industrial system, the performance improvement are due to several factors, including the edge-set representation with multi-modality, the data prefetch, and consolidation. In future, we will fully evaluate our system and extend the system onto distributed computing environment to address extremely large scale graphs.

References

[1] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1:1–1:20, 2016.

[2] K. Duraisamy, H. Lu, P. P. Pande, and A. Kalyanaraman, "High-performance and energy-efficient network-on-chip architectures for graph analytics," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 4, pp. 66:1–66:26, 2016.

[3] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "I/O efficient ECC graph decomposition via graph reduction," in *PVLDB*, 2016, pp. 516 – 527.

[4] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu, "Core decomposition in large temporal graphs," in *IEEE Big Data*, 2015, pp. 649–658.

[5] Y. Xia, I. G. Tanase, L. Nai, W. Tan, Y. G. Liu, J. Crawford, and C.-Y. Lin, "Explore efficient data organization for large scale graph analytics and storage," in *IEEE Big Data*, 2014, pp. 942 – 951.

[6] I. Filippidou and Y. Kotidis, "Online and on-demand partitioning of streaming graphs," in *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 4–13.

[7] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2012, pp. 31–46.

[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.