

G-Store: High-Performance Graph Store for Trillion-Edge Processing

Pradeep Kumar H. Howie Huang

Department of Electrical and Computer Engineering
George Washington University

Email: {pradeepk, howie}@gwu.edu

Abstract—High-performance graph processing brings great benefits to a wide range of scientific applications, e.g., biology networks, recommendation systems, and social networks, where such graphs have grown to terabytes of data with billions of vertices and trillions of edges. Subsequently, storage performance plays a critical role in designing a high-performance computer system for graph analytics. In this paper, we present *G-Store*, a new graph store that incorporates three techniques to accelerate the I/O and computation of graph algorithms. First, *G-Store* develops a space-efficient tile format for graph data, which takes advantage of the symmetry present in graphs as well as a new smallest number of bits representation. Second, *G-Store* utilizes tile-based physical grouping on disks so that multi-core CPUs can achieve high cache and memory performance and fully utilize the throughput from an array of solid-state disks. Third, *G-Store* employs a novel slide-cache-rewind strategy to pipeline graph I/O and computing. With a modest amount of memory, *G-Store* utilizes a proactive caching strategy in the system so that all fetched graph data are fully utilized before evicted from memory. We evaluate *G-Store* on a number of graphs against two state-of-the-art graph engines and show that *G-Store* achieves 2 to 8× saving in storage and outperforms both by 2 to 32×. *G-Store* is able to run different algorithms on trillion-edge graphs within tens of minutes, setting a new milestone in semi-external graph processing system.

I. INTRODUCTION

Graph processing has become increasingly important for our society with broad applications such as health sciences [18], web search [16], and social networks [6]. Technology advancements and its reach to masses have enabled such networks to cross the level of billions of vertices and trillions of edges, reaching terabytes of data [9].

A number of notable projects have developed cost-efficient graph computing systems that run on a single machine, e.g., GraphChi [20], X-Stream [28], TurboGraph [15], FlashGraph [39], GridGraph [40], and GraphQ [35], to name a few. In these systems, the performance of graph algorithms is closely tied to I/O throughput of the underlying storage system, as big graphs cannot fit entirely in the limited size of main memory.

Unfortunately, while prior work provides a variety of I/O optimizations to address the problem, much remains to be desired, especially the ability of high-performance processing of trillion-edge graphs. Such graphs pose new system level challenges not only in storage cost but also in memory and

cache management. In this work, we find out that current graph storage formats have several levels of redundancies that incur high I/O cost and thus unnecessarily lower algorithm performance. On the other hand, the main memory in existing systems is utilized, often with simple LRU policy, which is far from optimal for graph processing.

To address these challenges, we propose *G-Store*, a high-performance graph store which delivers high throughput of graph data I/Os from SSDs (solid-state drives), combined with judicious use of main memory and cache. The main contributions of *G-Store* are as follows:

First, *G-Store* utilizes the *symmetry* present in graph data by storing only the upper triangle (half) of graph data for undirected graphs. Similarly, it stores either in-edges or out-edges for directed graphs. Thus *G-Store* saves 2× storage and memory space for algorithm execution. More importantly, extending the traditional 2D partitioning approach, we further apply the technique of *smallest number of bits (SNB) representation* to store the graph in *tiles*, which eliminates redundant bits in vertex IDs and achieves another 4× space saving. In total, an 8× storage saving over traditional storage formats enables faster I/Os of graph data from SSDs, greatly improving the performance of graph algorithms.

Second, *G-Store* utilizes *tile-based physical grouping* on disks so that higher cache hit ratio on multi-core CPUs can be achieved for faster graph processing. In addition, *G-Store* uses asynchronous I/O (AIO) to load tiles in order to fully utilize the available bandwidth from an SSD array. This is done by batching data reads in fewer system calls using Linux AIO instead of direct and synchronous POSIX I/O.

Third, *G-Store* employs a *slide-cache-rewind (SCR)* strategy to pipeline graph I/O and computing. We propose tile-based in-memory *proactive caching* where *G-Store* is able to predict which tile will be required in the next iteration of graph processing with the help of algorithmic metadata. In addition, *G-Store* rewinds the order of the processing at the beginning of each iteration. These techniques utilize the data that is already fetched in memory, eliminating the occurrence of redundant I/Os in the future iterations.

We have conducted a number of experiments to show the advantage of *G-Store* and the impacts of different optimization techniques. *G-Store* outperforms X-Stream [28] and FlashGraph [39] by upto 32× and 2.4×, respectively. Further, *G-*

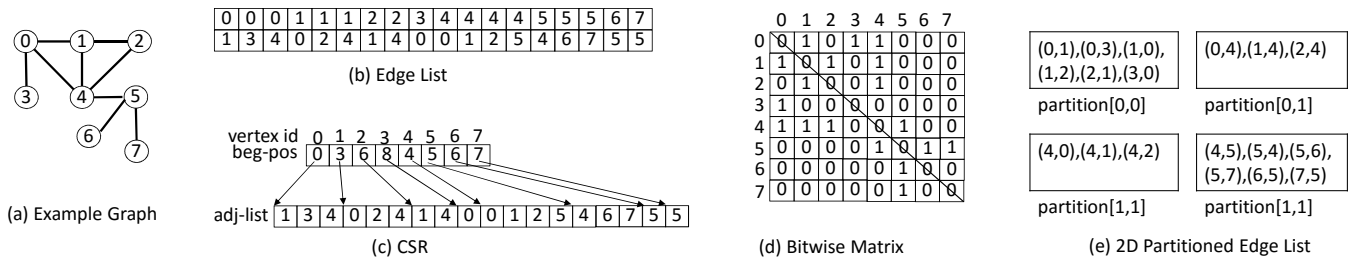


Fig. 1: Sample graph and its various representation format

Store is able to run different algorithms on trillion-edge graphs within tens of minutes.

The remainder of the paper is organized as follows. Section II presents the background on graph storage format and graph algorithms. Section III discusses the observations on which G-Store is based and presents an overview of G-Store. Section IV discusses our proposed graph representation techniques, and Section V discusses the physical grouping and on-disk layout of data. Section VI describes memory management, caching policy and thread utilization. Experimental results are presented in Section VII. It also presents the effect of various techniques on performance improvement. Section VIII presents related work and we conclude in Section IX.

II. BACKGROUND

A. Graph Representations

For a graph $G = (V, E)$, where V represents the set of the vertices and E the edges, there are a number of popular graph representations as follows:

Edge List representation is defined as a collection of tuples of vertices and each tuple represents a single edge. If two vertices v_i and v_j are connected then the edge tuple is represented as (v_i, v_j) . The size of the edge list representation of a graph equals to the product of the edge count and twice of the size of a vertex. Figure 1 shows an example graph (a) and its edge list representation (b).

Compressed Sparse Row (CSR) groups the edges of a vertex together and are stored in the *adjacency list array* (adj-list). There is a separate data structure of the *beginning position array* (beg-pos) that contains the index of the first edge for each vertex. The size of CSR representation of a graph equals the size of adjacency list ($|E|$), plus the size of beginning position array ($|V|$). Figure 1(c) shows the CSR representation of the example graph.

Matrix format consists of rows that represent the IDs of the source vertices of the edges and columns that represent the destination vertices. This is generally represented as the bitwise format where each row is a sequence of zeros and ones. The value of one represents an edge from the source to the destination vertex. Figure 1(d) shows the same graph in the bitwise matrix format.

2D Partitioned Edge List is a 2-level classification of the edge list format, where edge tuples are first partitioned using the ID of the source vertex followed by the ID of the destination vertex. Figure 1(e) shows the 2x2 partition of the sample graph,

where the edges are separated into different partitions based on vertex ID range 0-3 and 4-7.

Formally, we refer to each partition in the format of $\text{partition}[i,j]$, where i represents the partitions along the row major (X-axis) while j denotes those along the column major (Y-axis). For example, $\text{partition}[0,0]$ and $\text{partition}[0,1]$ constitute row[0] and $\text{partition}[0,1]$ and $\text{partition}[1,1]$ constitute column[0]. In this work, we propose several optimizations to 2D partitioning and use the term of *tile*¹ to represent a partition with such optimizations.

B. Graph Algorithms

In this paper we focus on three important graph algorithms that involve a variety of different random and sequential I/Os of the graph data. For each algorithm, we will also describe the *metadata* that are defined as the algorithm specific information maintained for a vertex or edge.

Breadth First Search (BFS) [5], [21], [22], is one of the most important graph algorithms as highlighted in Graph500 [14]. The BFS traversal starts from the root vertex and traverses each node by its depth thus covering vertices that are equidistant from the root vertex. The final output generates a tree showing how the vertices are connected in the original graph, so that the vertices can be reached in the minimum connection (called depth) from the root. The BFS graph traversal leads to random I/Os of the graph data. For power-law and small diameter graphs, BFS can be optimized for the explosion level when most of vertices are visited in this iteration [5]. BFS can also be implemented using the asynchronous method which reduces the total number of iterations needed [26].

PageRank (PR) [6] is used to rank the vertices in a graph by the order of their popularity. A vertex is more popular if its followers (the vertices that affect this node through incoming edges) are popular. In the iterative procedure of determining the page rank, the popularity of each vertex is transmitted to its neighbors, by dividing its popularity by the out-degree of the vertex. The iteration continues till the rank of each vertex stabilizes which is determined by the difference between the old and new rank value falling within a small range as decided by the programmers. The graph is accessed sequentially in a shared-memory programming model. Note that the metadata access can be random in this algorithm, e.g., in the PageRank computation of each vertex, the page rank values of its neighbors (randomly distributed) need to be accessed.

¹We use the name of *tiles* to emphasize the reduction of redundancy in storage, while *tiles* and *partitions* can be used interchangeably in the paper.

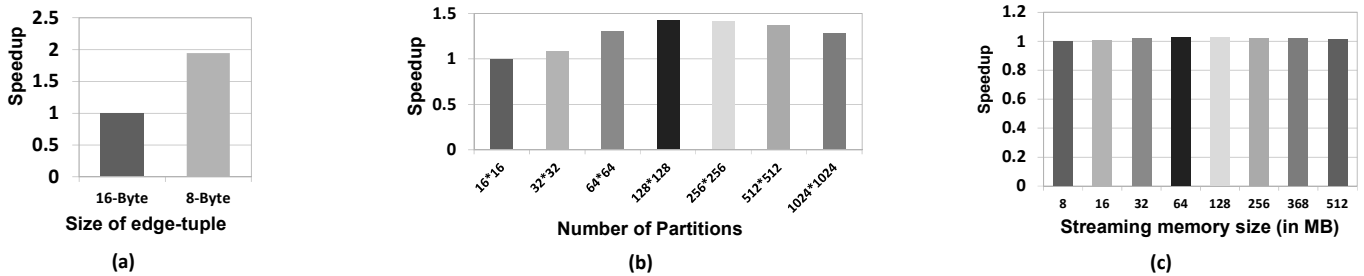


Fig. 2: Effect on PageRank performance: (a) edge tuple size; (b) metadata access localization; and (c) streaming memory size

Connected Component (CC) is a subgraph of the original graph where each vertex is connected to all other vertices of the subgraph through a direct edge or a path spanning the vertices of this subgraph. For the directed graph, the connected component could be strongly connected component (SCC) if a directed path or edge exists between all pairs of vertices of the subgraph. Alternatively, it is called weakly connected component (WCC) if replacing directed edges with undirected edges changes the subgraph to connected. Prior work [31] proposes an efficient parallel algorithm to identify different CC and is extended in [4]. The advantage of using this method is that all CCs are identified in very few iterations where all iterations are run in the whole graph taking advantage of sequential bandwidth.

It should be noted that the above three graph algorithms can be categorized in two classes: algorithms that process all the vertices in an iterative manner (PageRank, Connected Components) and those that have one or multiple starting points (anchored computations, e.g., BFS). In this paper, we use these three algorithms to motivate and demonstrate the effectiveness of G-Store.

III. G-STORE OVERVIEW

In this section, we present the architecture of G-Store, a high-performance graph store optimized for semi-external graph processing. To improve the performance of graph algorithms, prior work mainly aims to leverage sequential I/Os from the storage systems, e.g., GraphChi proposes a change in the representation of graph data to enable sequential reads. Similarly, X-Stream follows a scatter-gather-apply programming model where updates are generated and stored in disk first and then they are read and applied. Again, this programming model enables sequential read of graph data and updates, as well as sequential updates. On the other hand, FlashGraph and TurboGraph allow parallel I/O requests to saturate SSDs that provide a good support of random I/Os, and also implement selective reads of graph data to reduce the number of I/Os.

Different from prior approaches, G-Store not only focuses on maximizing the I/O performance on the graph data but also aims to improve the access to the metadata in the graph algorithms. In particular, G-Store explores four key aspects of I/O design: storage format, on-disk layout, I/O, and memory management. Figure 3 presents the overview of G-Store.

In the following we present three observations that motivate our design.

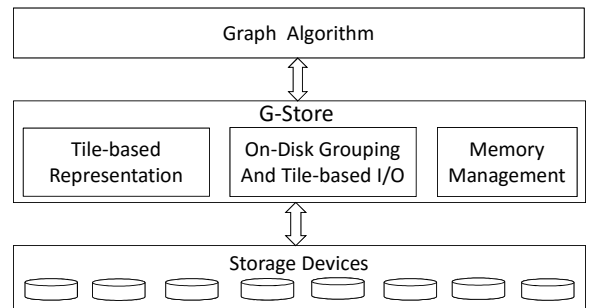


Fig. 3: G-Store framework

Observation 1: There is a need for a space-efficient graph storage format that can greatly reduce the I/O time, leading to an improved algorithmic performance. This is because when the graph data utilizes less disk storage, loading it to memory naturally requires a small number of I/Os. To illustrate this point, one may allocate 8 or 16 bytes to represent the edge tuple for a Kronecker graph (scale 28 and edge factor 16, Kron-28-16) [14]. When running X-Stream, the PageRank algorithm, which loads the graph from disk to memory in each iteration, doubles the performance when the storage space of the same graph is halved, as shown in Figure 2(a).

In this paper, we propose a new 2D tile-based storage format that leverages the symmetry and smallest number of bits representation for upto $8\times$ storage saving compared to traditional edge based graph data. This format will be utilized to accelerate the I/O process for different graph algorithms.

Observation 2: Localization of the metadata access is desired to scale the performance to multiple disks. As more disks are available in a system, the processing must support the increased throughput coming out of aggregation of disks. At this point, it is critical that the working set can fit in the CPU cache. Again, using PageRank as an example, calculating the page rank of a vertex will need the ranks of its neighbors. This task can be very fast if the ranks of its neighbors were in cache, or much slower otherwise. Figure 2(b) shows the variation of an in-memory PageRank algorithm, depending on whether one may localize the access of the neighbors' page rank values. With a 2D partitioning of the graph data, the performance of the PageRank algorithm improves when there are around 128 to 256 partitions for a Kronecker graph.

In this work, G-Store utilizes the 2D partitioning a step further and explores tile-based physical grouping in conjunction with the last-level cache (LLC) so that multi-core CPUs

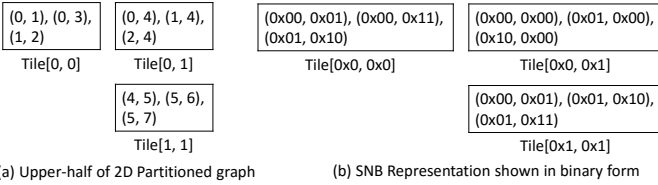


Fig. 4: Graph data after different saving techniques for the example graph in Figure 1.

can work in tandem with the increased throughput of the SSD array. This idea is to group the smaller tiles into bigger groups on disk so that the LLC can be utilized fully before the graph data are evicted.

Observation 3: As streaming the graph data requires a small memory footprint, there exists an opportunity for improving algorithmic performance through effective caching in the main memory. Figure 2(c) shows that the amount of the memory reserved for data streaming has very limited effect on performance. Clearly, as the graph algorithm is disk-bounded, allocating more memory will unlikely improve the disk throughput or performance. Unfortunately, prior work has paid little attention about caching the graph data. X-Stream simply streams the graph data without any caching. Although FlashGraph employs the LRU technique, the likelihood of the same data being used in the same iteration is negligible after the graph data has been used in this iteration.

In this paper, we explore how the graph data will be used in the next iteration and define appropriate caching strategies. This varies for different algorithms. In BFS, the cached data may never be utilized in later iterations as the adjacency list of a previously visited node will never need to be accessed again. On the other hand, for PageRank, all of the graph data would be utilized for the next iteration. Hence, G-Store employs a dynamic approach of proactive caching where G-Store is able to predict which tile will be required in the next iteration. Furthermore, G-Store uses a slide-cache-rewind (SCR) scheduler to pipeline I/O and processing, and to provide judicious usage of the main memory. In particular, G-Store rewinds at the end of each iteration and utilizes the data that is present in the memory before eviction, avoiding expensive redundant disk I/Os in the future.

IV. TILE-BASED STORAGE FOR GRAPHS

The proposed tile-based graph storage in G-Store is derived from 2D partitioning of the graph data. Specifically, it consists of two components: exploiting graph symmetry and leveraging offsets in the 2D grid representation.

A. Symmetry-based Storage Saving

Undirected Graph: Generally, an undirected graph is represented by storing each edge twice. That is, an edge $(v1, v2)$ is stored twice as $(v1, v2)$ and $(v2, v1)$. In Figure 1(e), one can see that in 2D partitioned graph data, partition[0,1] and partition[1,0] are a duplicate of each other. Even the partitions along the diagonal, i.e., partition[0,0] and [1,1], can be divided into two parts, where one contains the mirror image of the other.

To avoid the redundancy, G-Store stores and uses only the upper triangle of the 2D partitioned graph instead of the

Algorithm 1 BFS on the partition[i,j] of undirected graph

```

1:  $edge \leftarrow get\_edge\_ptr(i, j)$ ;
2: for  $k \leftarrow 1, edge\_count(i, j)$  do
3:    $src \leftarrow edge[k].src$ ;
4:    $dst \leftarrow edge[k].dst$ ;
5:   if  $depth[src] == level \ \& \ depth[dst] == INF$  then
6:      $depth[dst] \leftarrow level + 1$ ;
7:   end if
8:   // Added code for new storage format
9:   if  $depth[dst] == level \ \& \ depth[src] == INF$  then
10:     $depth[src] \leftarrow level + 1$ ;
11:  end if
12: end for

```

complete data set. As shown in Figure 4(a), one only needs three tiles, each of which has three edges. Tile[1,0] is no longer required in this case.

With small modifications, the graph algorithms can easily be adapted to work on only the upper triangle. Algorithm 1 shows the pseudocode for BFS on the partition[i,j] of 2D partitioned undirected graph. Lines numbered 8-10 are the new codes that have been added in BFS in order to work with the half of the graph data. We have developed codes in G-Store to support different graph algorithms.

Directed Graph: Each edge in directed graphs has a direction associated with it. An edge is called *in-edge* for a vertex if the edge is directed to this vertex otherwise it is called *out-edge*. It should be noted that an out-edge of one vertex would be an in-edge of another vertex. Correspondingly, each vertex also has *in-degree* and *out-degree*. The number of edges directed to a vertex is called its in-degree while the number of edges coming from this vertex and directed to any other vertex called out-degree. As a result, the implementation of graph algorithms in directed graphs differ from their undirected counterparts.

In G-Store, we believe that *storing and later loading both the in-edges and out-edges are redundant and unnecessary for graph processing in directed graph*. Note that many existing graph engines [20], [39] that store and load in-edges and out-edges both for directed graphs. We will show that compared to G-Store, these systems would require additional time to load the data.

Clearly, the graph algorithms will require small modifications in order to work with either in-edges or out-edges of directed graphs. Algorithm 2 shows the revised implementation of label propagation for the CC algorithm [39], where G-Store does not need to broadcast the label through out-edges. As a result, G-Store is able to cut the cost of data access in graph algorithm by half, while achieving the same results. Alternatively, this algorithm can be implemented using just out-edges as well.

In summary, for undirected graphs, enabling symmetry in the CSR format would incur high I/O cost for some algorithms (e.g., BFS) which is likely the reason why no existing graph engine supports it, and for directed graphs, the utilization of symmetry is not possible for many algorithms (e.g., SCC [10]) which need both in-edges and out-edges. The novelty of G-Store lies in addressing both problems with tile-based storage.

Algorithm 2 Label Propagation Using In-Edges

```
1: procedure UPDATELABEL(Vertex)
2:   min_label  $\leftarrow$  Vertex.label;
3:   for e in Vertex.inEdges() do
4:     min_label  $\leftarrow$  min(e.neighborLabel, min_label);
5:   end for
6:   Vertex.label  $\leftarrow$  min_label;
7:   // No need to broadcast
8:   // for e in Vertex.outEdges();
9:   // e.neighbourLabel  $\leftarrow$  vertex.label
10:  // end for
11: end procedure
```

B. Smallest Number of Bits Representation

This technique is based on the observation that another type of redundancy exists in the graph data, that is, *in a 2D partitioned graph, for all the edge tuples within a partition, the most-significant-bits (MSBs) of IDs of source and destination vertices are identical and hence redundant.*

For the example graph in Figure 1(a), normally a three-bit storage would be required to represent any vertex whose ID is from zero to seven, thus six bits in total for an edge tuple. Interestingly, fewer number of bits will be required, if one separates the representation of a partition and the edges in this partition. For example, for each edge tuple in tile[1,1], the MSBs of source and destination vertices are always [1,1]. Similarly, the MSBs of source and destination vertices of each edge tuple in tile[1,0] are always [1,0]. In this example shown in Figure 4(b), one may use two bits to represent one vertex and thus four bits for an edge tuple, saving two bits per edge tuple. Of course, there is additional storage (two bits in this case) needed to represent the IDs of the tiles, but the total storage cost is much smaller.

To calculate the original edge tuple from this compact representation, one must know to which tile an edge tuple belongs. This can be easily done by concatenating the tile ID to the vertex ID. In the example, tile[1,1] has the offset of (4,4), and the edge tuple (0,1) in this tile will represent the edge (4,5) in the original graph data. In G-Store, we calculate and cache the offset pointers of algorithmic metadata for each tile, so that any accesses to the metadata for the whole tile can be simply indexed with the compact vertex IDs. For each metadata, two operations will be needed per tile: one serves the compact source vertices and another compact destination vertices.

Tile Sizes: In this work, we allocate two bytes to represent each vertex and four bytes for an edge tuple. In tile[*i*,*j*], the range of the source vertices is [$i * 2^{16}, (i + 1) * 2^{16}$) and the destination vertices [$j * 2^{16}, (j + 1) * 2^{16}$). In another words, each tile can have at most 2^{32} edges and takes at most 16GB storage on disks. Note that the size of tiles varies by the graphs, and some are very sparse as small as a few bytes.

Figure 5 shows the variation in the edge count in individual tiles for the Twitter graph. We will present all the graphs used in this paper in Section VII. For Twitter, 40% of the tiles have no edges, while 82% with less than 1,000 edges and only 0.2% with more than 100,000 edges. The biggest tile has over 36 million edges. In contrast, most (98%) tiles for the synthetic

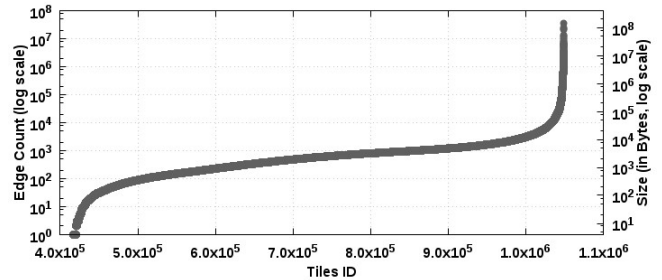


Fig. 5: Edge counts and size of Twitter tiles. Tile IDs are sorted by edge counts.

Kron-28-16 graph have less than 1,000 edges, with 4,097 edges in the biggest tile. On the other hand, the total number of the tiles that a graph has depends upon the number of vertices. For example, the Kron-28-16 graph (undirected) would have 8 million tiles with 256 million vertices, while Twitter (directed) has 1 million tiles with over 52 million vertices.

By just allocating four bytes to represent an edge tuple, we save a lot of space on memory and disk, which allows *G-Store* to scale to big graphs. In comparison, the existing work using edge tuple representation [28] and 2D partitioned graph [40], allocates 8 bytes for graphs with less than 2^{32} vertices, 16 for graphs with more than 2^{32} vertices, and so on.

Implementation: We store all the tiles in a single file. We do not allocate individual files for each tile as the number of files would be enormous and put unnecessary burden on storage. To track the size of each tile, we store the starting edge number of every tile in another array and store that as a separate file called *start-edge* file. This file serves similar purpose as does the *beg-pos* for the CSR format. For undirected graphs, the *start-edge* file also stores the information of just a half of the graph.

For conversion, we need two passes over the traditional edge tuple format in order to convert it to the tile. In the first pass, the start-edge array is calculated. In the second pass, edge tuples are converted to the space-efficient tile representation and stored at the right location. The method is similar to CSR conversion process. To this end, we find that conversion to G-Store format is faster than CSR in most cases. For example, the tile-based format needs 57 seconds for a Kron-28-16 graph, compared to 89 seconds for the CSR format. Table I shows the conversion time. Twitter conversion is slower because there is more skewness in the edge distribution in tiles in this case. For the Twitter graph, 40% tiles are empty, and 36 million edges are in one tile where the largest degree of a vertex is 779,958.

TABLE I: Conversion Time (in seconds)

Graph	Kron-28-16	Twitter	Friendster	Subdomain
CSR	89	16	26	17
G-Store	57	25	9	3

C. Additional Storage Saving on Degrees

Many graph algorithms need data on vertex degrees. For a power-law graph, the degrees of the majority of vertices

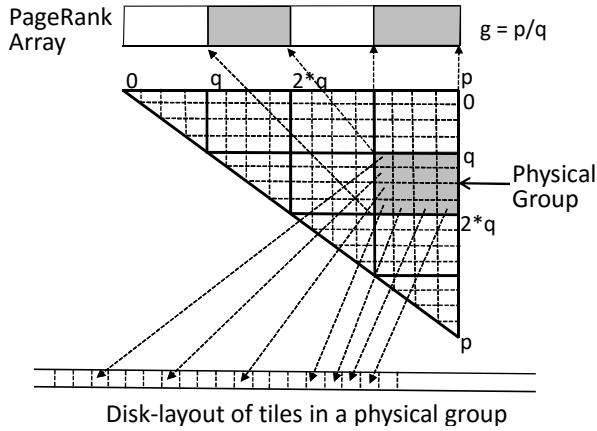


Fig. 6: On-disk Layout and algorithmic metadata (PageRank) access-localization for a physical group of tiles

are very small while some vertices have very high degree. Consequently, allocating the same number of bytes to store the degree would waste lots of space, yet another type of redundancy in storage.

In G-Store, we allocate two bytes to represent the degree (upto 32,767) of each vertices with the MSB set to zero. For vertices whose degree is greater than 32,767, the MSB bit is set one and the rest of the bit contains an index into another array which stores the degree. This additional optimization can further reduce the storage requirement of the most power-law graph, e.g., the size of degree array comes down from 4GB to 2GB for the Kron-30-16 graph. This space optimization can only be applied when the number of large degree vertices are less than 32,767.

In summary, graph symmetry provides $2\times$ storage saving while tile-based storage format offers additional 2 to $4\times$ saving. For instance, a 33-scale Kronecker file with edge factor 16 would require 4TB disk space in traditional 2D partition or edge tuple list, while *G-Store* needs only 512GB disk space for graph data, with additional 65GB for the start-edge file.

V. TILE GROUPING AND ASYNCHRONOUS I/O

A. On-Disk Grouping

As the graph size increases, the algorithmic metadata may not fit in CPU caches. The net effect is a drop in performance due to many cache misses. Prior work [39] shows that not all graph algorithms can saturate the disk I/O in a multi-SSD system. To address this problem, it is imperative that we utilize the last-level cache efficiently in order to sustain the maximum I/O throughput of the SSD array. This provides great opportunity for performance optimization for graph processing.

When a tile is processed in G-Store, the access to the corresponding metadata will be limited to the vertices within the range of the tile. For the Twitter graph, the metadata sizes of one tile are 64KB, 256KB, and 256KB for BFS (depth array), WCC (component id array), and PageRank (pagerank array), respectively. Though these data structures can fit within the available L2 cache, the LLCs have become much bigger in recent years, e.g., the latest Intel processors have up to 40MB

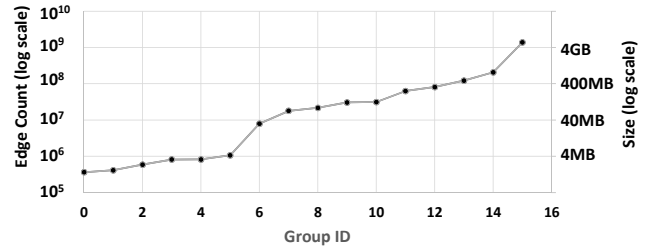


Fig. 7: Range of edge counts in Twitter physical groups. Group IDs are sorted by edge counts.

LLCs [17]. To take advantage of this feature, G-Store pushes the idea of tile-based partition one step further, by grouping tiles into physical groups as shown in Figure 6. This grouping is done at the disk storage level, i.e., the graph data is laid out in the disk in a way such that all the data of one physical group can be read sequentially. Moreover, the algorithmic metadata for one physical group fits in LLC for fast access.

Assuming a graph has p^2 tiles, each *group* takes q^2 tiles, depending on the size of LLC. Thus, there will be $g = p/q$ number of such groups. With physical groups, we can fetch the data and metadata for each group, process it and remove it at the end of the process. Next, we will proceed to the next group and load its data and metadata and process it. All this happens in a pipelined fashion as we will present in next section.

Figure 7 shows the edge count and size of a physical-group for the Twitter graph when q is equal to 256. There are 364,227 edges in the smallest group while over a billion edges are present in the largest group, that is, mostly tens to hundreds of MBs in size.

B. Tile-based I/O

We choose individual tile as a unit to manage the graph data, hence we do not fetch, process or cache partial data from any tile.

As tiles are grouped together sequentially in a physical-group, I/O happens for all the tiles in physical group before going to the second group, i.e, starting from the first physical group $[0, 0]$ fetching all the tiles from this group, followed by the physical group $[0,1]$, and so on. Within each group, G-Store starts fetching the data from the first tile and proceeds towards the last tile. Tiles from consecutive groups can be fetched together if the streaming memory size allows it.

G-Store uses Linux Asynchronous I/O (AIO) to fetch the graph data from disk, which consists of two steps, submitting the request to an I/O context and polling the context for completed I/O events. However, as the size of I/O increases, Linux AIO may not always show the asynchronous behavior if submitting the I/O blocks would take longer than polling. To this end, we use another property of Linux AIO to simplify the I/O interface. Linux AIO APIs (libaio) provides a single system call which can be used to batch many I/Os. At the runtime, G-Store identifies which tiles should be loaded based on the algorithmic metadata and does selective fetching. For example, in some of the last iterations of BFS very few vertices are frontiers, thus only few tiles may be required to fetch. In

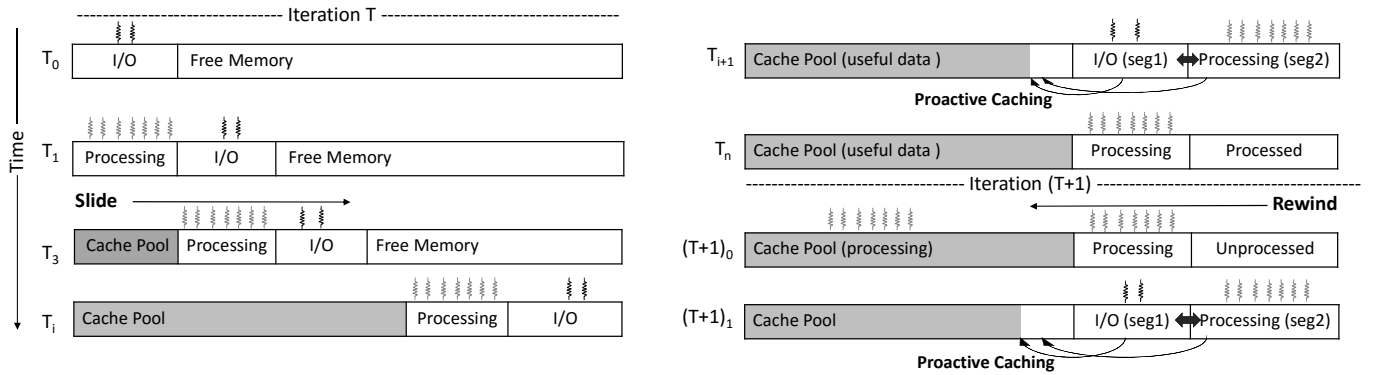


Fig. 8: SCR and memory management for streaming and caching the graph data. At time T_0 of the current iteration, G-Store fetches the first segment for the first iteration. At T_1 , the second segment is getting data from disk while the first segment is getting processed. At T_2 , the first segment is preserved, the second segment is getting processed while the third segment is fetching data from disk. At T_i , all memory is about to get exhausted, and at T_{i+1} , cache pool data is analyzed to free space. The last two segment will alternately be used for I/O and processing and useful data will be copied to the cache pool. At T_n , there is no I/O at the end of the current iteration. At time $(T+1)_0$ of the next iteration, there is the Rewind processing - the cache pool and the first segment are processed. At $(T+1)_1$, the second segment is processed, while the first segment is used for I/O. The cache pool is analyzed. Similar steps of I/O and processing will continue.

this case, these I/Os would be merged into a single AIO system call, which makes the management of I/O very easy.

In addition, G-Store uses Linux AIO for fetching the data in association with direct I/O to avoid the double copy, where the data is directly copied from disk to the userspace buffer skipping the kernel buffer. We will discuss the management of the userspace memory buffers in the next section.

VI. SCR: MEMORY AND I/O MANAGEMENT

In this section we will present how G-Store manages graph data and metadata in memory and the SCR technique that consists of three steps, slide, cache, and rewind.

A. Memory Layout

G-Store divides the whole memory into two parts: one for graph metadata (e.g., pagerank array) and the other for streaming and caching the graph data. Recall our observation that extra memory dedicated for streaming the graph data provides limited benefit, G-Store allocates the minimal memory for data streaming while the rest is used for caching data in an intelligent way.

There are two ways to manage the memory for caching. First, one can manage memory in terms of pages such as in Linux page cache. Linux AIO is aligned at 512 bytes, so the page size could be any multiple of this size. However, doing so requires lots of memory in page management metadata such as page header, while a large page size may lead to waste of memory in fragmentation as different tiles may not be exactly a multiplier of the page size.

Alternatively, G-Store utilizes copy-based memory management. The idea is to provide the optimum use of available memory without wasting a lot of memory in page-management, similar to [29]. In this method, available memory for graph data is dedicated for two fixed sized chunks called *segment*. The rest of the memory is allocated to the *cache pool* for graph data caching. Two segments are used for overlapping the graph I/O and processing, that is, one segment is used for

loading the data while the other is used for processing the previously loaded data.

Once I/O and processing finish the segment, the naive approach is to switch the role of them, that is, processing the I/O segment with most recently loaded data, and fetching data into the other segment. In this work, when the processing is done, G-Store instead puts the recently processed segment into the cache pool and allocates a new segment for I/O. This way, G-Store is able to save the data in the pool that most likely will be needed in the next iteration of graph processing.

B. Slide: Overlapping I/O and Processing

G-Store manages the thread scheduling, I/O and processing efficiently. It fetches the data on one segment and executes the program on the other segment which has been just fetched. These two operations are done in parallel using different threads available in a multi-core system. There will be many I/O and process phases in any iteration as the segment size is generally very small compared to graph data.

G-Store "slides" down the segments and memory, and alternates between I/O and processing. We achieve the parallelism in graph processing by assigning different rows to different threads. We use dynamic scheduling of OpenMP for workload balancing as there are many rows and the size of each row varies a lot.

Figure 8 shows an example of memory and I/O process for two consecutive iterations T and $T+1$, each of which consists of multiple steps represented by subscripts. Let us start with the free memory allocated for streaming and caching to understand the G-Store operation. At the first time interval T_0 of the iteration T , the first memory segment is allocated for I/O. At the next time interval T_1 , I/O is complete and the segment is sent for processing, and at the same time, another memory segment is allocated for I/O. At T_2 , when the processing of the first segment finishes, G-Store will manage it as the cache pool, while the same steps of I/O and processing are repeated as in previous time intervals.

The slide process goes on till G-Store runs out of the memory, which is illustrated as the time interval T_i . At this point, there are one contiguous cache pool and two segments (*seg1* and *seg2*) for I/O and processing, while no new unused segments are available. Because not everything in the cache pool is useful, G-Store uses proactive caching policy to decide which tiles to keep in the pool. This step reclaims memory from the pool and will be discussed shortly. The freed memory at the end of cache pool is used for caching additional graph data that would be fetched in later phases as shown in phase T_{i+1} . Moving tiles between memory segments are implemented with *memcpy* and *memmove* instructions.

The current iteration ends at the time interval T_n when G-Store does not fetch any data and just processes the last fetched segment. Also, G-Store does not analyze the segments for caching as they will be used shortly in next iteration when G-Store rewinds.

C. Proactive Caching

For many graph algorithms, data fetched in any iteration may not be used in the same or future iteration. In a 2D partitioned graph data, a tile may be needed again in the next iteration, e.g., when the tile generates new frontiers during the current processing of the BFS algorithm. Instead of simple LRU-like caching, G-Store designs a set of *proactive caching* rules based on algorithmic metadata to calculate which tiles need to be processed in the next iteration. This way, G-Store is able to fully utilize in-memory data and avoid redundant I/Os for the same data in the future. The proactive caching rules are slightly different for undirected and directed graphs. We summarize them for undirected graphs as follows:

Rule 1: At the end of the processing of any row[i], one shall know whether row[i] would be processed in the next iteration or not. For example, in Figure 1(c), vertices in the range 0-3 are processed only in row[0] (i.e., tile[0,0] and tile[0,1]), while vertices in the range 4-7 are processed in row[1] plus column[1]. Thus any frontier in the range 0-3 can only be generated during the processing of row[0] and no new frontiers in this range can be generated in later processing.

Rule 2: If row[i] is not needed for the next iteration, then one shall know that tile[i,i] will not be needed. There exists partial information on whether other individual tiles, i.e., from tile [i, i+1] to tile[i, p-1] would be required or not for the next iteration. This knowledge about tile[i, i+k] would be completely available only when we process row[i+k]. For example, if row[0] is not needed in the next iteration then one will know whether there is a need for tile[0,1] when we process row[1].

For directed graphs, we store only out-edges. The rule is simple: at the end of processing any row[i], we partially know which of the row would be processed again in the next iteration. So, if row[i] is potentially needed in the next iteration, it will be cached.

As a result, at the end of the processing of row[i], we know either fully or partially if this row is needed for next iteration. However, the cache analysis happens only when the cache pool is full, i.e., at time T_i in the Figure 8. This means that we have accumulated more data in cache pool, and more information in

the form of algorithmic metadata before any tiles are evicted. This helps us in utilizing the cache capacity to cache only the required tiles. Even if some tiles are evicted because of partial information available till now, the free space in the cache pool is again utilized for future reads of this iteration as shown at time T_{i+1} in the Figure 8.

D. Rewind

In the beginning of each iteration, G-Store holds a large amount of graph data in the cache pool that has been fetched and processed in the prior iteration. Interestingly, for many graph algorithms from PageRank to WCC, almost 100% of these data will be utilized in the current iterations, which G-Store manages to save thanks to the proactive caching policy.

To take advantage of cached data, G-Store "rewinds" the processing in between two iterations to process the data in the cache pool, as well as the data present in two segments that have been fetched in the last two phases of previous iteration. An example can be found at the first time interval $(T+1)_0$ in Figure 8, when G-Store rewinds and processes the entire data in the cache pool. At this time, no I/O is performed. In the next interval $(T+1)_1$, G-Store starts to load data into the free segment and continues to slide and cache.

The handling of the last two segments, *seg1* and *seg2*, is slightly complicated. Specifically, in the beginning of the iteration $T+1$, G-Store processes *seg1* first and does not fetch any data during this time. Note that it also processes the cache pool at the same time. In the next interval, it processes the second segment *seg2* while starting to load *seg1* with graph data.

The rewind process not only reduces the number of I/Os by reusing the data that is already cached in the memory, but also reveals the hints for future caching. Specifically, new information (such as new frontiers in BFS) are available during the beginning of an iteration after processing the cache pool and two segments. Using this information to cache the data would be more beneficial for the next iteration. In addition, since the cache is also analyzed, G-Store will evict the data that may not be needed in the next iteration after processing, thus freeing up some space for further caching.

VII. EXPERIMENTS

The machine used for the experiments has a dual-socket of Intel Xeon CPU E5-2683 2GHz, each having 14 cores (total 56 threads with hyper-threading). The CPU has 32K data and instruction L1 cache, 256K L2 and 16M L3 (LLC) cache. For storage, the machine uses eight SAMSUNG 850 EVO 512GB SSDs connected to LSI SAS9300-8i HBA configured in software RAID-0. The OS is Centos 7.1 with Linux kernel 3.10. The collection of graphs is shown in Table II. We will use these graphs to demonstrate the effectiveness of the design choices that we have made and to compare the performance against the existing work. In the tests, we use 8GB memory for streaming and caching data (4GB for the Twitter, Friendster and Subdomain graphs due to their smaller data sizes). The segment size is set to 256MB.

TABLE II: Different graphs and their sizes

Graph Name	Graph Type	Vertex Count	Edge Count	Edge List Size	CSR Size	G-Store Size	Space Saving w.r.t. Edge List	Space Saving w.r.t. CSR
Twitter [2]	(Un-)Directed	52579682	1963263821	14.6GB	14.6GB	7.3GB	2x	2x
Friendster [1]	(Un-)Directed	68349466	2586147869	19.26GB	19.26GB	9.63GB	2x	2x
Subdomain [3]	(Un-)Directed	101717775	2043203933	15.22GB	15.22GB	7.6 GB	2x	2x
Rmat-28-16	Undirected	2^{28}	2^{33}	64GB	32GB	16GB	4x	2x
Random-27-32	Undirected	2^{27}	2^{33}	64GB	32GB	16GB	4x	2x
Kron-28-16	Undirected	2^{28}	2^{33}	64GB	32GB	16GB	4x	2x
Kron-30-16	Undirected	2^{30}	2^{35}	256GB	128GB	64GB	4x	2x
Kron-33-16	Undirected	2^{33}	2^{38}	4TB	2TB	512GB	8x	4x
Kron-31-256	Undirected	2^{31}	2^{40}	8TB	4TB	2TB	4x	2x

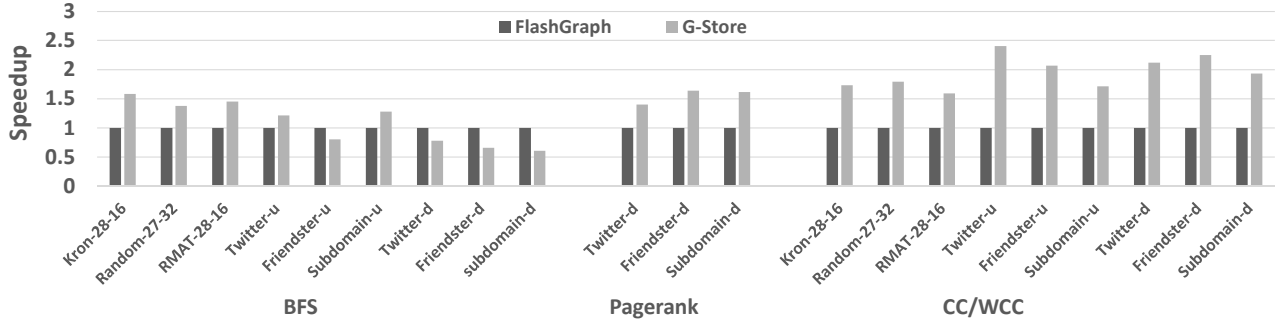


Fig. 9: Speedup comparison of G-Store with FlashGraph. The symbol -u stands for undirected graphs and -d directed graphs.

A. Trillion-Edge Processing

We are able to achieve fast graph processing in minutes for Kron-31-256 which has two billion vertices and one trillion edges. The total memory usage including all the metadata is about 14GB for BFS, 36GB for PageRank and 20GB for WCC, which also accounts for the 8GB memory reserved for data streaming and caching.

The runtime for this graph is presented in Table III. G-Store calculates WCC in 32 minutes and traverses this large graph under 43 minutes. That is, G-Store achieves external BFS performance of 432 MTEPS (million traversed edges per second). Also, G-Store is able to perform one iteration of PageRank in this graph in 14 minutes using only one machine. In contrast, for such an iteration, prior work [9] needs about 3 minutes on 200 machines on a trillion-edge graph with less number of vertices (1.39 billion vertices).

TABLE III: Runtime (in seconds) of trillion-edge graphs

Graph	BFS	PageRank	WCC
Kron-31-256	2548.546	4214.543	1925.134
Kron-33-16	1509.13	1882.88	849.046

We also process Kron-33-16 graph which has 8 billion vertices and 256 billion edges where a vertex ID needs 8 bytes of storage. In this case, G-Store needs no extra tuning for processing while other solutions, e.g., [39] [28] [40], would require to change the definition of vertex ID from 32-bit to 64-bit, doubling the cost on memory and storage.

B. G-Store vs External Memory Graph Systems

We compare G-Store with FlashGraph and X-Stream for all three graph algorithms and various undirected and directed graphs. Like G-Store, FlashGraph is a semi-external graph engine while X-Stream is fully external platform. Besides using memory for data streaming, FlashGraph and G-Store need to allocate additional memory for graph and algorithmic metadata. FlashGraph implements a different flavor of PageRank [38] where they send only the delta of most recent PageRank update to neighbors. It should be noted that FlashGraph runs BFS and PageRank only on out-edges in the CSR format (no symmetry advantage) and hence, G-Store does not have any space saving for directed graphs (with less than 2^{32} vertices) for BFS and PageRank. FlashGraph do not provide a PageRank implementation for undirected graphs.

Overall, compared to X-Stream, G-Store achieves more than $17\times$, $21\times$ and $32\times$ improvement in BFS, Pagerank and CC respectively for the Kron-28-16 graph. The speedup for the Twitter graph are $12\times$ (BFS), $9\times$ (PageRank) and $17\times$ (CC). We have also observed similar speedups for other graphs.

Figure 9 shows the relative speedup of G-Store over FlashGraph performance. G-Store achieves $2\times$ and $1.5\times$ speedup for PageRank and CC. In particular, for CC, G-Store is more than twice as fast as FlashGraph on both Twitter and Friendster, regardless of directed or undirected edges. For BFS, it outperforms (average $1.4\times$) on undirected graphs, with the only exception of Friendster graph where G-Store reads slightly more data in last several iterations. For directed graphs, G-Store is slightly worse (20% for Twitter) due to no space saving benefit.

C. Impact of Different Optimizations

Space Saving: The space saving from Symmetry and SNB representation can be observed in Table II. For graphs with less than 4 billion vertices we achieve 4× space saving with respect to X-Stream and 2× compared to FlashGraph. For larger graphs, we achieve 8× space saving compared to X-Stream and 4× with respect to FlashGraph. In this case, G-Store uses two bytes for a vertex ID, compared to eight bytes in prior approaches. The space-saving helps in achieving huge performance improvement and enables some graph data to fit within the available memory.

We compare the performance when we store all the graph data without any space saving, using symmetry, and when using symmetry and SNB both for the Kron-28-16 graph. We allocate 8GB of memory in the above three cases. Figure 10 shows the speedup. Utilizing the symmetry in the Kron-28-16 graph doubles the speed, while SNB improves the performance further to 4.9× for BFS and 4.8× for PageRank. The speedup is more than 4× (the space-saving factor) because G-Store is able to cache more data and process at a faster speed.

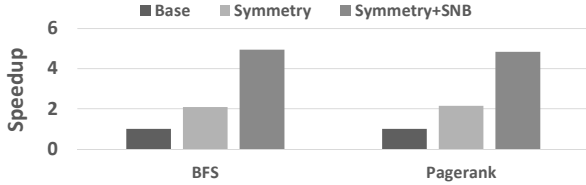


Fig. 10: Speedup due to space saving

Physical Group: We choose PageRank to evaluate the effectiveness of physical grouping because it is compute intensive and any optimization related to hardware cache utilization would be most visible. Figure 11 shows the relative in-memory performance when different number of tiles are grouped together for the Kron-30-16 graph. We find that grouping 256x256 tiles performs 57% better compared to 32x32 grouping. In this case, one can see that LLC transaction (Load/Store) and miss count are both at the minimum. Figure 12 shows that there are upto 21% reduction in number of transactions and 35% in misses over other scenarios. This combination is ideal for the best performance and can give us best scaling to multiple disks.

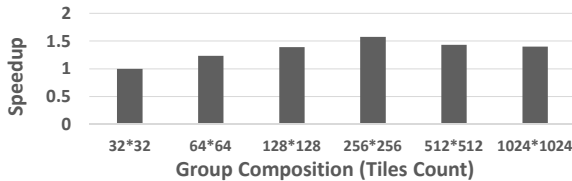


Fig. 11: In-memory speedup from grouping

Slide-Cache-Rewind: We study the effectiveness of SCR technique with respect to a base policy where we have just two segments of memory to fetch and overlap it with processing. We allocate 8GB of memory and use 256MB for each segment while the remaining 7.5GB is used for caching. In the baseline case, we allocate two segments for 4GB size each. The fetching and processing of segments are completely overlapped for

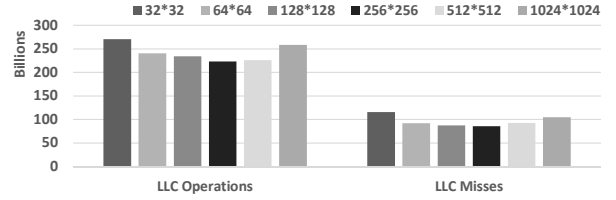


Fig. 12: LLC operations and misses for various grouping sizes

PageRank. But for BFS, we fetch for the next iteration only when we finish processing the current iteration, because at this point the full information about which tiles to fetch is not completely available. Figure 13 shows the speedup in performance of BFS and PageRank for the Kron-28-16 graph. The design choice is able to provide over 60% performance improvement for BFS and over 35% for PageRank and WCC.

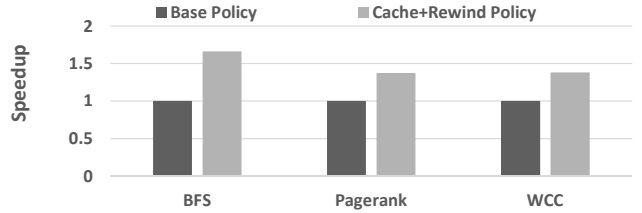


Fig. 13: Speedup from caching and scheduling policy

Cache Size: In this experiment, 256MB is allocated to each segment while the total memory size is varied from 1GB to 8GB in steps for the Kron-28-16 graph. For the Twitter graph, the memory size is varied from 1GB to 4GB. Further increasing memory size will make the graph fit completely in memory. Figure 14 shows the speedup in both cases. Over 30% performance improvement is achieved for the Kron-28-16 graph with 8GB memory. For Twitter, we observe the maximum 46% improvement for BFS, 39% for PageRank, and 37% for WCC, respectively.

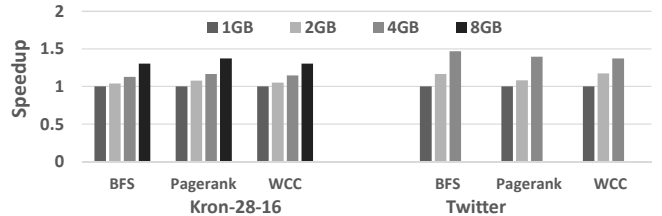


Fig. 14: Effect of different cache sizes

D. Scalability on SSDs

We use Linux software RAID0 to bundle the disks together with the stripe size set to 64KB. Figure 15 shows that G-Store is able to achieve close to ideal 4× speedup on 4 disks and around 6× on 8 disks for the Kron-30-16 graph. This shows that G-Store can deliver graph data at a high throughput that saturates the CPU cores. This is in particular clear for PageRank as it saturates the CPU well before it can saturate the combined I/O throughput of 8 SSDs.

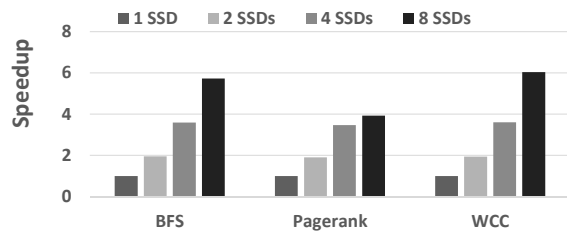


Fig. 15: Scalability on SSDs

VIII. RELATED WORK

There have been a number of external graph engines such as GraphChi [20] and X-Stream [28], which are optimized for sequential performance of hard drives, as well as TurboGraph [15], which is an external graph system designed for SSDs. The most related project is FlashGraph [39], which also processes graph on SSDs in the semi-external manner. Our key improvements come from utilizing space-efficient tile-based storage, on-disk layout, and slide-cache-rewind I/O technique. Further, GridGraph [40] also uses a 2D partitioning scheme to achieve better performance and selective I/O in a single machine setup. While GridGraph depends upon Linux page-cache for caching, G-Store exploits the properties of 2D tiles to cache data that are most likely to be needed in the next iteration.

PathGraph [37] is another semi-external graph engine and Ligra+ [33] is an in-memory graph engine. Both propose to use delta based compression, which requires that the data must be sorted. Gbase [19] is a map-reduce based graph management system that also uses compression techniques. In G-Store, the smallest number of bits representation provides an space efficient way to represent graph in a 2D partition. Compression can be applied to the data present in tiles to provide further space saving, which we leave as future work. On the other hand, Galois [25] and Ligra [32] are highly optimized in-memory graph processing engine. G-Store can take advantage of such algorithmic techniques to have an efficient in-memory processing.

Physical grouping of G-Store is inspired from locality-aware data placement [34], [40] to utilize the LLC fully. Our tiles representation helps in utilizing the smaller L2 caches too. In addition, a number of graph engines such as PowerGraph [12], GraphLab [23], Pregel [24], Giraph [11], PowerLyra [8], GBase [19], Trinity [30], and GraphX [13] have been proposed to process the graph in a distributed environment. For example, PowerGraph shows that an efficient partitioning can reduce the communication between different distributed machines. Chaos [27], an scale-out graph processing engine points out that the scale-36 RMAT graph with a trillion of edges would need 16TB of storage space. Prior work [7], [36] uses 2D partitioning to divide the data among many nodes in supercomputers. In contrast, G-Store utilizes 2D partitioning with the new tile-based storage for space efficiency, which we envision can be easily extended to a distributed system.

IX. CONCLUSION AND FUTURE WORK

In this paper, we have shown that an space-efficient representation provides a better I/O performance in a external graph

engine. In G-Store, complete overlapping of I/O and compute, and optimum utilization of hardware cache further help in scaling the solution to multiple disks. We have also shown how the available memory can be utilized in a better way for caching the graph data. In summary, G-Store leverages these techniques to run different graph algorithms very efficiently on an array of SSDs and is able to quickly process trillion-edge graphs. For future work, we plan to extend G-Store to support even larger graphs on a tiered storage, where SSDs can be utilized with a set of hard drives, and deploy in large-scale computing systems.

ACKNOWLEDGMENTS

We would like to thank the SC reviewers for their helpful suggestions. This work is supported in part by NSF CAREER award 1350766 and grants 1124813 and 1618706.

REFERENCES

- [1] Friendster Network Dataset – KONECT. <http://konect.uni-koblenz.de/networks/friendster>.
- [2] Twitter (MPI) Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter_mpi.
- [3] Web Graphs. <http://webdatacommons.org/hyperlinkgraph/2012-08/download.html>.
- [4] D. A. Bader, G. Cong, and J. Feo. On the Architectural Requirements for Efficient Execution of Graph Algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing*, 2005.
- [5] S. Beamer, K. Asanovic, and D. Patterson. Direction-Optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.
- [6] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, Amsterdam, The Netherlands, 1998.
- [7] F. Checconi and F. Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014.
- [8] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.
- [9] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-scale. *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Aug. 2015.
- [10] L. K. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*. 2000.
- [11] Giraph. <http://giraph.apache.org>.
- [12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.
- [14] Graph500. <http://www.graph500.org/>.
- [15] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. 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*, 2013.
- [16] B. A. Huberman and L. A. Adamic. Internet: Growth dynamics of the World-Wide Web. *Nature*, 1999.
- [17] Intel Haswell. [https://en.wikipedia.org/wiki/Haswell_\(microarchitecture\)](https://en.wikipedia.org/wiki/Haswell_(microarchitecture)).

- [18] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 2000.
- [19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: A Scalable and General Graph Management System. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.
- [20] A. Kyrola, G. E. Blleloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [21] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.
- [22] H. Liu, H. H. Huang, and Y. Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2016.
- [23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment (VLDB)*, 2012.
- [24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the SIGMOD International Conference on Management of data*, 2010.
- [25] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
- [26] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, 2010.
- [27] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.
- [28] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
- [29] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.
- [30] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2013.
- [31] Y. Shiloach and U. Vishkin. An $O(\log n)$ Parallel Connectivity Algorithm. *Journal of Algorithms*, 1982.
- [32] J. Shun and G. E. Blleloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.
- [33] J. Shun, L. Dhulipala, and G. E. Blleloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *Proceedings of the 2015 Data Compression Conference (DCC)*, 2015.
- [34] J. Wang, Q. Xiao, J. Yin, and P. Shang. DRAW: A New Data-gRouping-AWare Data Placement Scheme for Data Intensive Applications With Interest Locality. *IEEE Transactions on Magnetics*, June 2013.
- [35] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the Usenix Annual Technical Conference*, 2015.
- [36] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Nov 2005.
- [37] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.
- [38] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, Aug 2014.
- [39] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. 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 (FAST)*, 2015.
- [40] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.