GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning

Xiaowei Zhu, Wentao Han, and Wenguang Chen

Department of Computer Science and Technology, Tsinghua University

{zhuxw13,hwt04}@mails.tsinghua.edu.cn, cwg@tsinghua.edu.cn

Abstract

In this paper, we present GridGraph, a system for processing large-scale graphs on a single machine. GridGraph breaks graphs into 1D-partitioned vertex chunks and 2D-partitioned edge blocks using a first fine-grained level partitioning in preprocessing. A second coarse-grained level partitioning is applied in runtime. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reduce the I/O amount required for computation. The partitioning of edges also enable selective scheduling so that some of the blocks can be skipped to reduce unnecessary I/O. This is very effective when the active vertex set shrinks with convergence.

Our evaluation results show that GridGraph scales seamlessly with memory capacity and disk bandwidth, and outperforms state-of-the-art out-of-core systems, including GraphChi and X-Stream. Furthermore, we show that the performance of GridGraph is even competitive with distributed systems, and it also provides significant cost efficiency in cloud environment.

1 Introduction

There has been increasing interests to process large-scale graphs efficiently in both academic and industrial communities. Many real-world problems, such as online social networks, web graphs, user-item matrices, and more, can be represented as graph computing problems.

Many distributed graph processing systems like Pregel [17], GraphLab [16], PowerGraph [8], GraphX [28], and others [1, 22] have been proposed in the past few years. They are able to handle graphs of very large scale by exploiting the powerful computation resources of clusters. However, load imbalance [11, 20], synchronization overhead [33] and fault tolerance overhead [27] are still challenges for graph processing in distributed environment. Moreover, users need to be skillful since tuning a cluster and optimizing graph algorithms in distributed systems are non-trivial tasks.

GraphChi [13], X-Stream [21] and other out-of-core systems [9, 15, 31, 34] provide alternative solutions. They enable users to process large-scale graphs on a single machine by using disks efficiently. GraphChi partitions the vertices into disjoint intervals and breaks the large edge list into smaller shards containing edges with destinations in corresponding intervals. It uses a vertex-centric processing model, which gathers data from neighbors by reading edge values, computes and applies new values to vertices, and scatters new data to neighbors by writing values on edges. By using a novel parallel sliding windows method to reduce random I/O accesses, GraphChi is able to process large-scale graphs in reasonable time. However, its sharding process requires the edges in every shard to be sorted by sources. This is a very time consuming process since several passes over edges are needed. Fragmented accesses over several shards are often inevitable, decreasing the usage of disk bandwidth. X-Stream introduces an edge-centric scatter-gather processing model. In the scatter phase, X-Stream streams every edge and generates updates to propagate vertex states. In the gather phase, X-Stream streams every update, and applies it to the corresponding vertex state. Accesses to vertices are random and happen on a high level of storage hierarchy which is small but fast. And accesses to edges and updates fall into a low level of storage hierarchy which is large but slow. However, these accesses are sequential so that maximum throughput can be achieved. Although X-Stream can leverage high disk bandwidth by sequential accessing, it needs to generate updates which could be in the same magnitude as edges, and its lack of support on selective scheduling could also be a critical problem when dealing with graphs of large diameters.

We propose GridGraph, which groups edges into a "grid" representation. In GridGraph, vertices are partitioned into 1D chunks and edges are partitioned into 2D

blocks according to the source and destination vertices. We apply a logical higher level of partitioning in runtime. Chunks and blocks are grouped into larger ones for I/O efficiency. Different from current vertex-centric or edge-centric processing model, GridGraph combines the scatter and gather phases into one "streaming-apply" phase, which streams every edge and applies the generated update instantly onto the source or destination vertex. By aggregating the updates, only one pass over the edge blocks is needed. This is nearly optimal for iterative global computation, and is suited to both in-memory and out-of-core situations. Moreover, GridGraph offers selective scheduling, so that streaming on unnecessary edges can be reduced. This significantly improves performance for many iterative algorithms.

We evaluate GridGraph on real-world graphs and show that GridGraph outperforms state-of-the-art out-of-core graph engines by up to an order of magnitude. We also show that GridGraph has competitive performance to distributed solutions, and is far more cost effective and convenient to use. In summary, the contributions of this paper are:

- A novel grid representation for graphs, which can be generated from the original edge list using a fast range-based partitioning method. Different from the index plus adjacency list or shard representation that requires sorting, edge blocks of the grid can be transformed from an unsorted edge list with small preprocessing overhead, and can be utilized for different algorithms and on different machines.
- A 2-level hierarchical partitioning schema, which is effective for not only out-of-core but also in-memory scenarios.
- A fast streaming-apply graph processing model, which optimizes I/O amount. The locality of vertex accesses is guaranteed by dual sliding windows. Moreover, only one sequential read of edges is needed and the write amount is optimized to the order of vertices instead of edges.
- A flexible programming interface consisting of two streaming functions for vertices and edges respectively. Programmers can specify an optional user-defined filter function to skip computation on inactive vertices or edges. This improves performance significantly for iterative algorithms that active vertex set shrinks with convergence.

The remaining part of this paper is organized as follows. Section 2 describes the grid representation, which is at the core of GridGraph. Section 3 presents the computation model, and the 2-level hierarchical partitioning schema. Section 4 evaluates the system, and compares it with state-of-the-art systems. Section 5 discusses related works, and finally Section 6 concludes the paper.

2 Graph Representation

The grid representation plays an important role in GridGraph. We introduce the details of the grid format, as well as how the fast partitioning process works. We also make a comparison with other out-of-core graph engines and discuss the trade-offs in preprocessing.

2.1 The Grid Format

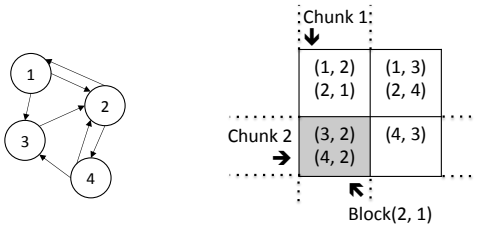
Partitioning is employed to process a graph larger than the memory capacity of a single machine. GraphChi designs the shard data format, and stores the adjacency list in several shards so that each shard can be fit into memory. Vertices are divided into disjoint intervals. Each interval associates a shard, which stores all the edges with destination vertex in the interval. Inside each shard, edges are sorted by source vertex and combined into the compact adjacency format. X-Stream also divides the vertices into disjoint subsets. A streaming partition consists of a vertex set, an edge list and an update list, so that data of each vertex set can be fit into memory. The edge list of a streaming partition (in the scatter phase) consists of all edges whose source vertex is in the partition's vertex set. The update list of a streaming partition (in the gather phase) consists of all updates whose destination vertex is in the partition's vertex set.

GridGraph partitions the vertices into P equalized vertex chunks. Each chunk contains vertices within a contiguous range. The whole $P \times P$ blocks can be viewed as a grid, and each edge is put into a corresponding block using the following rule: the source vertex determines the row of the block and the destination vertex determines the column of the block. Figure 1(b) illustrates how GridGraph partitions the example graph in Figure 1(a). There are 4 vertices in this graph and we choose $P = 2$ for this example. $\{1, 2\}$ and $\{3, 4\}$ are the 2 vertex chunks. For example, Edge (3, 2) is partitioned to Block (2, 1) since Vertex 3 belongs to Chunk 2 and Vertex 1 belongs to Chunk 1.

In addition to the edge blocks, GridGraph creates a metadata file which contains global information of the represented graph, including the number of vertices V and edges E, the partition count P, and the edge type T (to indicate whether the edges are weighted or not). Each edge block corresponds to a file on disks.

GridGraph does preprocessing in the following way:

1. The main thread sequentially reads edges from original unordered edge list and divides them into batches of edges and pushes each batch to the task



(a) An example graph (b) Grid representation

Figure 1: Organization of the edge blocks

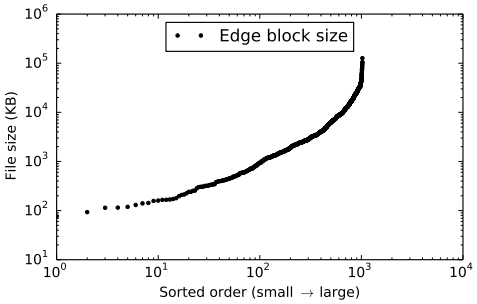


Figure 2: Edge block size distribution of Twitter graph using a 32×32 partitioning.

queue (to achieve substantial sequential disk bandwidth, we choose 24MB to be the size of each edge batch).

2. Each worker thread fetches a task from the queue, calculates the block that each edge in this batch belongs to, and appends the edge to the corresponding edge block file. To improve I/O throughput, each worker thread maintains a local buffer of each block, and flushes to files once the buffer is full.

After the partitioning process, GridGraph is ready to do computation. However, due to the irregular structure of real world graphs, some edge blocks might be too small to achieve substantial sequential bandwidth on HDDs. Figure 2 shows the distribution of edge block sizes in Twitter [12] graph using a 32×32 partitioning, which conforms to the power-law [7], with a large number of small files and a few big ones. Thus full sequential bandwidth can not be achieved sometimes due to potentially frequent disk seeks. To avoid such performance loss, an extra merge phase is required for GridGraph to perform better on HDD based systems, in which the edge block files are appended into a large file one by one and the start offset of each block is recorded in metadata. The time taken by each phase is shown in Section 4.

2.2 Discussion

Different from the shard representation used in GraphChi, GridGraph does not require the edges in each block to be sorted. This hence reduces both I/O and computation overhead in preprocessing. We only need to read and write the edges from and to disks once, rather than several passes over the edges in GraphChi. This lightweight preprocessing procedure can be finished very quickly (see Table 2), which is much faster than the preprocessing of GraphChi.

X-Stream, on the other hand, does not require explicit preprocessing. Edges are shuffled to several files according to the streaming partition. No sorting is required and the number of partitions is quite small. For many graphs that all the vertex data can be fit into memory, only one streaming partitions is needed. However, this partitioning strategy makes it inefficient for selective scheduling, which can largely affect its performance on many iterative algorithms that only a portion of the vertices are used in some iterations.

It takes very short time for GridGraph to complete the preprocessing. Moreover, the generated grid format can be utilized in all algorithms running on the same graph. By partitioning, GridGraph is able to conduct selective scheduling and reduce uncessary accesses to edge blocks without active edges[1]. We can see that this contributes a lot in many iterative algorithms like BFS and WCC (see Section 4), which a large portion of vertices are inactive in many iterations.

The selection of the number of partitions P is very important. With a more fine-grained partitioning (which means a larger value of P), while the preprocessing time becomes longer, better access locality of vertex data and more potential in selective scheduling can be achieved. Thus a larger P is preferred in partitioning. Currently, we choose P in such a way that the vertex data can be fit into last level cache. We choose P to be the minimum integer such that

$$\frac{V}{P} \times U \leq C,$$

where C is the size of last level cache and U is the data size of each vertex. This partitioning shows not only good performance (especially for in-memory situations) but also reasonable preprocessing cost. In Section 4, we evaluate the impact of P and discuss the trade-offs inside.

3 The Streaming-Apply Processing Model

GridGraph uses a streaming-apply processing model in which only one (read-only) pass over the edges is required and the write I/O amount is optimized to one pass over the vertices.

[1] An edge is active if its source vertex is active.

3.1 Programming Abstraction

GridGraph provides 2 functions to stream over vertices (Algorithm 1) and edges (Algorithm 2).

Algorithm 1 Vertex Streaming Interface

```
function StreamVertices(F_v, F)
    Sum = 0
    for each vertex do
        if F(vertex) then
            Sum += F_v(edge)
        end if
    end for
    return Sum
end function
```

Algorithm 2 Edge Streaming Interface

```
function StreamEdges(F_e, F)
    Sum = 0
    for each active block do          ▷ block with active edges
        for each edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
    end for
    return Sum
end function
```

F is an optional user defined function which accepts a vertex as input and should returns a boolean value to indicate whether the vertex is needed in streaming. It is used when the algorithm needs selective scheduling to skip some useless streaming and is often used together with a bitmap, which can express the active vertex set compactly. F_e and F_v are user defined functions which describe the behavior of streaming. They accept an edge (for F_e), or a vertex (for F_v) as input, and should return a value of type R. The return values are accumulated and as the final reduced result to user. This value is often used to get the number of activated vertices, but is not restricted to this usage, e.g. users can use this function to get the sum of differences between iterations in PageRank to decide whether to stop computation.

GridGraph stores vertex data on disks. Each vertex data file corresponds to a vertex data vector. We use the memory mapping mechanism to reference vertex data backed in files. It provides convenient and transparent access to vectors, and simplifies the programming model: developers can treat it as normal arrays just as if they are in memory.

We use PageRank [19] as an example to show how to implement algorithms using GridGraph (shown in Algorithm 3[2]). PageRank is a link analysis algorithm that calculates a numerical weighting to each vertex in the graph to measure its relative importance among the vertices. The PR value of each vertex is initialized to 1. In each iteration, each vertex sends out their contributions to neighbors, which is the current PR value divided by its out degree. Each vertex sums up the contributions collected from neighbors and sets it as the new PR value. It converges when the mean difference reaches some threshold[3].

Algorithm 3 PageRank

```
function Contribute(e)
    Accum(&NewPR[e.dest], PR[e.source] / Deg[e.source])
end function
function Compute(v)
    NewPR[v] = 1 − d + d × NewPR[v]
    return |NewPR[v] − PR[v]|
end function
d = 0.85
PR = {1, ..., 1}
Converged = 0
while ¬Converged do
    NewPR = {0, ..., 0}
    StreamEdges(Contribute)
    Diff = StreamVertices(Compute)
    Swap(PR, NewPR)
    Converged = Diff / V ≤ Threshold
end while
```

3.2 Dual Sliding Windows

GridGraph streams edges block by block. When streaming a specific block, say, the block in the i-th row and j-th column, vertex data associated with this block falls into the i-th and j-th chunks. By selecting P such that each chunk is small enough to fit into the fast storage (i.e. memory for out-of-core situations or last level cache for in-memory situations), we can ensure good locality when accessing vertex data associated with the block being streamed.

The access sequence of blocks can be row-oriented or column-oriented, based on the update pattern. Assume that a vertex state is propagated from the source vertex to the destination vertex (which is the typical pattern in a lot of applications), i.e. source vertex data is read and destination vertex data is written. Since the column of each edge block corresponds to the destination vertex chunk, column oriented access order is preferred in this case. The destination vertex chunk is cached in memory when GridGraph streams blocks in the same column from top to bottom, so that expensive disk write operations are aggregated and minimized. This property is very impor-

[2] Accum(&s, a) is an atomic operation which adds a to s.

[3] Sometimes we run PageRank for certain number of iterations to analyze performance.

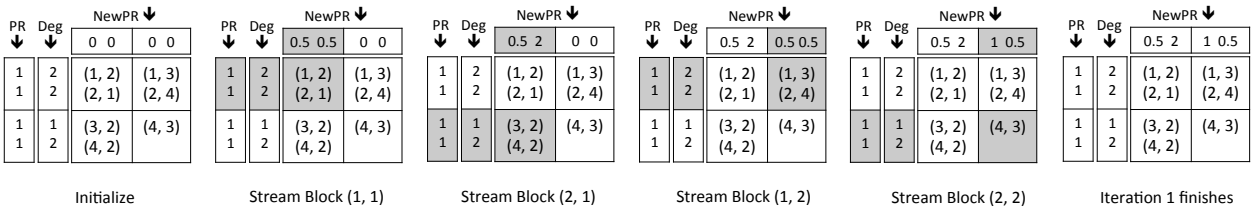


Figure 3: Illustration of dual sliding windows. It shows the first iteration of PageRank on the example graph. Shaded block and chunks are active (loaded into memory).

tant in practical use, especially for SSDs, since the write performance might degrade after writing large volume of data due to the write amplification phenomenon. On the other hand, since SSDs have upper limits of write cycles, it is thus important to reduce disk writes as much as possible to achieve ideal endurance.

Figure 3 visualizes how GridGraph uses dual sliding windows to apply updates onto vertices. PageRank is used as the example. The read window (in which we read current PageRank value and out degree from the source vertex) and the write window (in which we accumulate contributions to the new PageRank value of the destination vertex) slide as GridGraph streams edge blocks in the column-oriented order.

Since GridGraph applies in-place updates onto the active vertex chunk, there might exists data race, i.e. data of one vertex is concurrently modified by multiple worker threads. Thus, inside the process function F_e, users need to use atomic operations to apply thread-safe updates to vertices, so as to ensure the correctness of algorithms.

Utilizing the fact that the bandwidth of parallel randomized access to fast level storage is still orders of magnitude bigger than the sequential bandwidth of slow level storage (such as main memory vs. disks, and cache vs. main memory), the time of applying updates is overlapped with edge streaming. GridGraph only requires one read pass over the edges, which is advantageous to the solutions of GraphChi and X-Stream that need to mutate the edges (GraphChi) or first generating and then streaming through the updates (X-Stream).

Through read-only access to the edges, the memory required by GridGraph is very compact. In fact, it only needs a small buffer to hold the edge data being streamed so that other free memory can be used by page cache to hold edge data, which is very useful when active edge data becomes small enough to fit into memory.

Another advantage of this streaming-apply model is that it not only supports classical BSP [25] model, but also allows asynchronous [3] updates. Since vertex updates are in-place and instant, the effect of an update can be seen by following vertex accesses, which makes lots of iterative algorithms to converge faster.

3.3 2-Level Hierarchical Partitioning

We first give the I/O analysis of GridGraph in a complete streaming-apply iteration, which all the edges and vertices are accessed. Assume edge blocks are accessed in the column-oriented order. Edges are accessed once and source vertex data is read P times while destination vertex data is read and written once. Thus I/O amount is

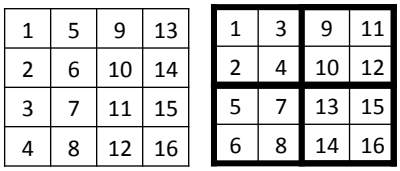
$$E + P \times V + 2 \times V$$

for each iteration. Thus a smaller P should be preferred to minimize I/O amount which seems opposite to the grid partitioning principle discussed in Section 2 that a larger P can ensure better locality and selective scheduling effect.

To deal with this dilemma, we apply a second level partitioning above the edge grid to reduce I/O accesses of vertices. The higher level partitioning consists of a $Q \times Q$ edge grid. Given a specified amount of memory M, and the size of each vertex data U (including source and destination vertex), Q is selected to be the minimum integer which satisfies the condition

$$\frac{V}{Q} \times U \leq M.$$

As we mentioned in Section 2, P is selected to fit vertex data into last level cache of which the capacity is much smaller than memory. Hence P is much bigger than Q, i.e. the $Q \times Q$ partitioning is more coarse-grained than the $P \times P$ one, so that we can just virtually group the edge blocks by adjusting the accessing orders of blocks. Figure 4 illustrates this concept. The preprocessed grid consists of 4×4 blocks, and a virtual 2×2 grid partitioning is applied over it. The whole grid is thus divided into 4 big blocks, with each big block containing 4 small blocks. The number inside each block indicates the access sequence. An exact column-oriented access order is used in the original 4×4 partitioning. After the second level 2×2 over 4×4 partitioning is applied, we access the coarse-grained (big) blocks in column-oriented order, and within each big block, we access the fine-grained (small) blocks in column-oriented order as well.



(a) 4x4 grid (b) 2x2 virtual grid

Figure 4: A 2-Level Hierarchical Partitioning Example. The number inside each block indicates the access sequence.

This 2-level hierarchical partitioning provides not only flexibility but also efficiency since the higher level partitioning is virtual and GridGraph is able to utilize the outcome of lower level partitioning thus no more actual overhead is added. At the same time, good properties of the original fine-grained edge grid such as more selective scheduling chances can still be leveraged.

3.4 Execution Implementation

Thanks to the good locality ensured by the property of dual sliding windows, the execution engine of GridGraph mainly concentrates on streaming edge blocks.

GridGraph streams each block sequentially. Before streaming, GridGraph first checks the activeness of each vertex chunk. Edge blocks are streamed one by one in the sequence that dual sliding windows needs, and if the corresponding source vertex chunk of the block is active, it is added to the task list.

GridGraph does computation as follows:

1. The main thread pushes tasks to the queue, containing the file, offset, and length to issue each read request. (Length is set to 24MB like in preprocessing to achieve full disk bandwidth.)

2. Worker thread fetches tasks from the queue until empty, read data from specified location and process each edge.

Each edge is first checked by a user defined filter function F, and if the source vertex is active, F_e is called on this edge to apply updates onto the source or destination vertex (note that we do not encourage users to apply updates onto both the source and destination vertex, which might make the memory mapped vector suffer from unexpected write backs onto the slow level storage).

4 Evaluation

We evaluate GridGraph on several real world social graphs and web graphs, and shows significant performance improvement compared with current out-of-core graph engines. GridGraph is even competitive with distributed systems when more powerful hardware can be utilized.

Dataset	V	E	Data size	P
LiveJournal	4.85M	69.0M	527 MB	4
Twitter	61.6M	1.47B	11 GB	32
UK	106M	3.74B	28 GB	64
Yahoo	1.41B	6.64B	50 GB	512

Table 2: Graph datasets used in evaluation.

4.1 Test Environment

Experiments are conducted on AWS EC2 storage optimized instances, including d2.xlarge instance, which contains 4 hyperthread vCPU cores, 30.5GB memory (30MB L3 Cache), and 3 HDDs of 2TB, and i2.xlarge instance, which has the same configuration with d2.xlarge except that it contains 1 800GB SSD instead of 3 2TB HDDs (and the L3 cache is 24MB). I2 instances are able to provide high IOPS while D2 instances can provide high-density storage. Both i2.xlarge and d2.xlarge instances can achieve more than 450GB/s sequential disk bandwidth.

For the I/O scalability evaluation, we also use more powerful i2.2xlarge, i2.4xlarge, and i2.8xlarge instances, which contain multiple (2, 4, 8) 800GB SSDs, as well as more (8, 16, 32) cores and (61GB, 122GB, 244GB) memory.

4.2 System Comparison

We evaluate the processing performance of GridGraph through comparison with the latest version of GraphChi and X-Stream on d2.xlarge[4] and i2.xlarge instances.

For each system, we run BFS, WCC, SpMV and Pagerank on 4 datasets: LiveJournal [2], Twitter [12], UK [4] and Yahoo [29]. All the graphs are real-world graphs with power-law degree distributions. LiveJournal and Twitter are social graphs, showing the following relationship between users within each online social network. UK and Yahoo are web graphs that consist of hyperlink relationships between web pages, with larger diameters than social graphs. Table 2 shows the magnitude, as well as our selection of P for each graph. For BFS and WCC, we run them until convergence, i.e. no more vertices can be found or updated; for SpMV, we run one iteration to calculate the multiplication result; and for PageRank, we run 20 iterations on each graph.

[4] A software RAID-0 array consisting of 3 HDDs is set up for evaluation on d2.xlarge instances.

	i2.xlarge (SSD)				d2.xlarge (HDD)			
	BFS	**WCC**	**SpMV**	**PageR.**	**BFS**	**WCC**	**SpMV**	**PageR.**
LiveJournal								
GraphChi	22.81	17.60	10.12	53.97	21.22	14.93	10.69	45.97
X-Stream	6.54	14.65	6.63	18.22	6.29	13.47	6.10	18.45
GridGraph	2.97	4.39	2.21	12.86	3.36	4.67	2.30	14.21
Twitter								
GraphChi	437.7	469.8	273.1	1263	443.3	406.1	220.7	1064
X-Stream	435.9	1199	143.9	1779	408.8	1089	128.3	1634
GridGraph	204.8	286.5	50.13	538.1	196.3	276.3	42.33	482.1
UK								
GraphChi	2768	1779	412.3	2083	3203	1709	401.2	2191
X-Stream	8081	12057	383.7	4374	7301	11066	319.4	4015
GridGraph	1843	1709	116.8	1347	1730	1609	97.38	1359
Yahoo								
GraphChi	-	114162	2676	13076	-	106735	3110	18361
X-Stream	-	-	1076	9957	-	-	1007	10575
GridGraph	16815	3602	263.1	4719	30178	4077	277.6	5118

Table 1: Execution time (in seconds) with 8 GB memory. "-" indicates that the corresponding system failed to finish execution in 48 hours.

GraphChi runs all algorithms in asynchronous mode, and an in-memory optimization is used when the number of vertices are small enough, so that vertex data can be allocated and hold in memory and thus edges are not modified during computation, which contributes a lot to performance on Twitter and UK graph.

Table 1 presents the performance of chosen algorithms on different graphs and systems with memory limited to 8GB to illustrate the applicability. Under this configuration, only the LiveJournal graph can be fit into memory, while other graphs require access to disks. We can see that GridGraph outperforms GraphChi and X-Stream on both HDD based d2.xlarge and SSD based i2.xlarge instances, and the performance does not vary much except for BFS on Yahoo, which lots of seek is experienced during the computation, thus making SSDs more advantageous than HDDs. In fact, sometimes better results can be achieved on d2.xlarge instance due to the fact that the peak sequential bandwidth of 3 HDDs on d2.xlarge is slightly greater than that of 1 SSD on i2.xlarge.

Figure 5 shows the disk bandwidth usage of 3 systems, which records the I/O throughput of a 10-minute interval running PageRank on Yahoo graph, using a d2.xlarge instance. X-Stream and GridGraph are available to exploit high sequential disk bandwidth while GraphChi is not so ideal due to more fragmented reads and writes across many shards. GridGraph try to minimize write amount thus more I/O is spent on read while X-Stream has to write a lot more data.

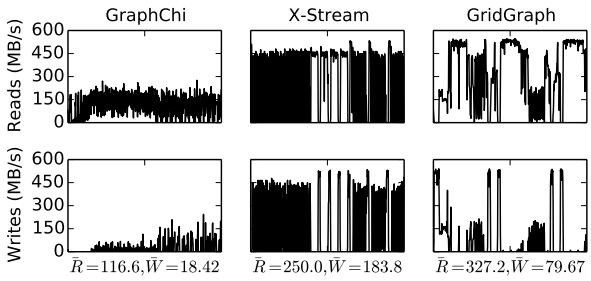
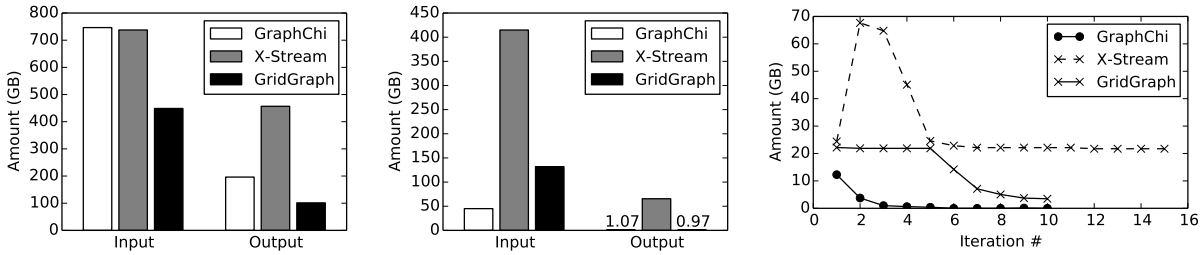


Figure 5: Disk bandwidth usage chart of a 10-minute interval on GraphChi, X-Stream and GridGraph. $\bar{R}$ = average read bandwidth; $\bar{W}$ = average write bandwidth.

For algorithms that all the vertices are active in computation, like SpMV and PageRank (thus every edge is streamed), GridGraph has significant reduction in I/O amount that is needed to complete computation. GraphChi needs to read from and write to edges to propagate vertex states, thus $2 \times E$ I/O amount to edges are required. X-Stream needs to read edges and generate updates in scatter phase, and read updates in gather phase, thus a total I/O amount of $3 \times E$ is required (note that the size of updates is in the same magnitude as edges). On the other hand, GridGraph only requires one read pass over the edges and several passes over the vertices. The write amount is also optimized in GridGraph, which



(a) PageRank on Yahoo graph (b) WCC on Twitter graph (c) WCC on Twitter graph

Figure 6: I/O amount comparison.

only one pass over the vertex data is needed. Figure 6(a) shows the I/O amount that each system needs to complete 5 iterations of PageRank on Yahoo graph. We can see that the input amount of GridGraph is about 60 percent of X-Stream and GraphChi, while the output amount of GridGraph is about 4.5 times less. Thanks to the nice vertex access pattern of dual sliding windows, vertex updates are aggregated within a chunk, which can be a very useful feature in practical use. For graphs that all the vertex data can be cached in memory (like in Figure 6(b)), only one disk read in the first iteration and one disk write after the final iteration is required no matter how many iterations are needed.

For iterative algorithms that only a portion of the whole vertex set participates in the computation of some iterations, such as BFS and WCC, GridGraph can benefit from another good property of grid partitioning that we can skip lots of useless reads with selective scheduling. We compare the I/O amount versus X-Stream and GraphChi using WCC on Twitter graph in Figure 6(b), and provide a per iteration I/O amount in Figure 6(c). We can see that I/O amount decreases with the convergence of the algorithm. In fact, when the volume of active edges reaches a certain level such that the data size is smaller than memory capacity, the page cache could buffer almost all of the edges that might be needed in latter iterations, thus complete the following computations very fast. This phenomenon is more obvious in web graphs than in social graphs, since the graph diameter is much bigger. GraphChi also supports selective scheduling, and its shard representation can have even better effect than GridGraph on I/O reduction. Though I/O amount required by GraphChi is rather small, it has to issue many fragmented reads across shards. Thus the performance is not ideal enough due to limited bandwidth usage.

Another interesting observation from Figure 6(c) is that GridGraph converges faster than X-Stream. This is due to the availability to support asynchronous update in GridGraph that applied updates can be used directly in the same iteration. In WCC, we always push the latest label through edges, which can speed up the label propagation process.

We conclude that GridGraph can perform well on large-scale real world graphs with limited resource. The reduction in I/O amount is the key to the performance gain.

4.3 Preprocessing Cost

Table 3 shows the preprocessing cost of GraphChi and GridGraph on i2.xlarge (SSD) and d2.xlarge (HDD) instances[5]. For SSDs, we only need to partition the edges and append them to different files. For HDDs, a merging phase that combines all the edge block files is required after partitioning. It is known that HDDs do not perform well on random access workloads due to high seek time and low I/O concurrency while SSDs do not have such severe performance penalty. As P increases, edge blocks become smaller and the number of blocks increases, thus making it hard for HDDs to achieve full sequential bandwidth since more time will be spent on seeking to potentially different positions. By merging all the blocks together and use offsets to indicate the region of each block, we can achieve full sequential throughput and benefit from selective scheduling at the same time.

We can see that GridGraph outperforms GraphChi in preprocessing time. GridGraph uses a lightweight range based partitioning, thus only one sequential read over the input edge list and append-only sequential writes to $P \times P$ edge block files are required, which can be handled very well by operating systems.

Figure 7 shows the preprocessing cost on a d2.xlarge instance using Twitter graph with different selections of P. We can see that as P becomes larger, the partitioning time and especially the merging time becomes longer, due to the fact that more small edge blocks are generated. Yet we can see the necessity of this merging phase on

[5]X-Stream does not require explicit preprocessing. Its preprocessing is covered in the first iteration before computation.

	C (S)	G (S)	C (H)	G (H) P	G (H) M	G (H) A
LiveJournal	14.73	1.99	13.06	1.64	1.02	2.66
Twitter	516.3	56.59	425.9	76.03	117.9	193.9
UK	1297	153.8	1084	167.6	329.7	497.3
Yahoo	2702	277.4	2913	352.5	2235.6	2588.1

Table 3: Preprocessing time (in seconds) for GraphChi and GridGraph on 4 datasets. C = GraphChi, G = GridGraph; S = SSD, H = HDD; P = time for partitioning phase, M = time for merging phase, A = overall time.

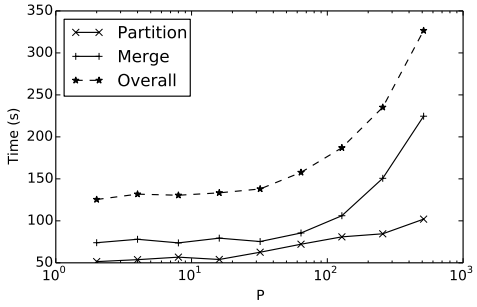


Figure 7: Preprocessing time on Twitter graph with different selections of P (from 2 to 512).

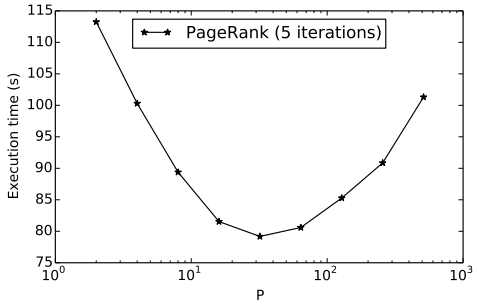


Figure 8: Execution time of PageRank on Twitter graph with different selections of P (from 2 to 512).

HDD based systems since several passes over the edge blocks are often required in multi-iteration computations, which can benefit a lot from this extra preprocessing cost.

We conclude that the preprocessing procedure in GridGraph is very efficient and essential.

4.4 Granularity of Partitioning

First, we evaluate the impact on performance with different selections of P for in-memory situations. Figure 8 shows the execution time of PageRank (5 iterations) on Twitter graph. We use all the available memory (30.5GB) of an i2.xlarge instance, so that the whole graph (including vertex and edge data) can be fit into memory. We can see that P should be neither too small nor too large to achieve a good performance. When P is too small, i.e. each vertex chunk can not be put into last level cache, the poor locality significantly affects the efficiency of vertex access. When P is too large, more data race between atomic operations in each vertex chunk also slows down the performance. Thus we should choose P by considering the size of last level cache as we mentioned in Section 2 to achieve good in-memory performance either when we have a big server with large memory or when we are trying to process a not so big graph that can be fit into memory.

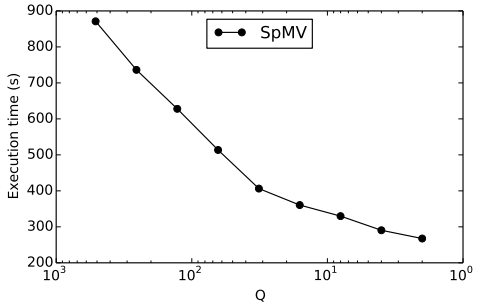


Figure 9: Execution time of SpMV on Yahoo graph with different selections of Q (from 512 to 2).

Next, we evaluate the impact of second level partitioning on performance for out-of-core scenarios. Figure 9 shows the execution time of SpMV on Yahoo graph with different selections of Q. Memory is limited to 8GB on an i2.xlarge instance so that the whole vertex data can not be cached in memory (via memory mapping). As Q decreases, we can see that the execution time descends dramatically due to the fact that a smaller Q can reduce the passes over source vertex data. Thus we should try to minimize Q when data of $\frac{V}{Q}$ vertices can be fit into memory, according to the analysis in Section 3.

We conclude that the 2-level hierarchical partitioning used in GridGraph is very essential to achieve good performance for both in-memory and out-of-core scenes.

Memory	Twitter WCC	Yahoo PageRank
8GB	286.5	1285
30.5GB	120.6	943.1

Table 4: Scalability with Memory. Execution time is in seconds.

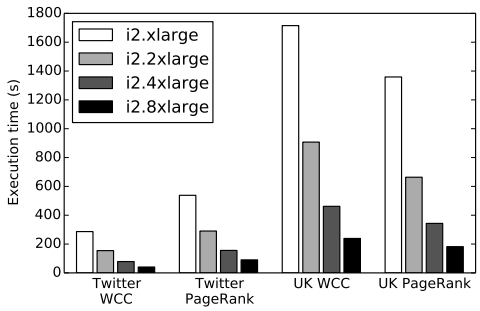


Figure 10: Scalability with I/O.

4.5 Scalability

We evaluate the scalability of GridGraph by observing the improvement when more hardware resource is added.

Table 4 shows the performance variance of WCC on Twitter graph and PageRank (5 iterations) on Yahoo graph when usable memory increases. With the memory increased from 8GB to 30.5GB, the whole undirected Twitter graph (22GB) can be fit into memory, so that edge data is read from disks only once. Meanwhile, with 30.5GB memory, the whole vertex data of Yahoo graph (16GB) can be cached in memory via memory mapping, thus only one pass of read (when program initializes) and write (when program exits) of vertex data is required.

We also evaluate the performance improvement with disk bandwidth. Figure 10 shows the performance when using other I2 instances. Each i2.[n]xlarge instance contains n 800GB SSDs, along with $4 \times n$ hyperthread vCPU cores, and $30.5 \times n$ RAM. Disks are set up as a software RAID-0 array. We do not limit the memory that GridGraph can use, but force direct I/O to the edges to bypass the effect of page cache. We can see that GridGraph scales almost linearly with disk bandwidth.

We conclude that GridGraph scales well when more powerful hardware resource can be utilized.

4.6 Comparison with Distributed Systems

From the result in Figure 10, we find that the performance of GridGraph is even competitive with distributed graph systems. Figure 11 shows the performance comparison between GridGraph using an i2.4xlarge instance (containing 16 hyperthread cores, 122GB RAM, 4 800GB SSDs, $3.41/h), versus PowerGraph and GraphX on a cluster with 16 m2.4xlarge instances (each with 8 cores, 68.4GB RAM, 2 840GB HDDs, $0.98/h), using the result from [28]. We can find that even single-node disk based solutions can provide good enough performance (note that we do edge streaming via direct I/O in Figure 10) and significant reduction in cost ($3.41/h vs. $15.68/h). Moreover, GridGraph is a very convenient single-node solution to use and deploy, thus reducing the effort that cluster-based solutions concerns about. In fact, limited scaling is observed in distributed graph engines ([28]) due to high communication overhead relative to computation in many graph algorithms while GridGraph can scale smoothly as the memory and I/O bandwidth being increased.

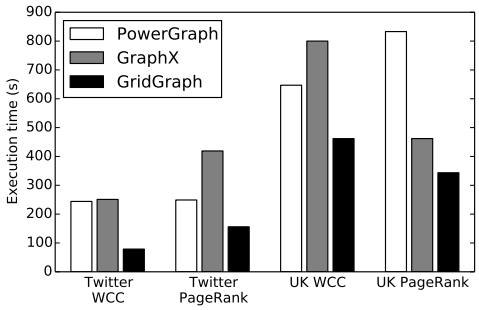


Figure 11: Performance comparison with PowerGraph and GraphX.

We conclude that GridGraph is competitive even with distributed systems when more powerful hardware is available.

5 Related Work

While we have discussed GraphChi and X-Stream in detail, there are other out-of-core graph engines using alternative approaches. TurboGraph [9] manages adjacency lists in pages, issues on-demand I/O requests to disks, and employs a cache manager to maintain frequently used pages in memory to reduce disk I/O. It is efficient for targeted queries like BFS while for applications that require global updates, the performance might degrade due to frequent accesses to the large vector backed on disks. FlashGraph [34] implements a semi-external memory graph engine which stores vertex states in memory and adjacency lists on SSDs, and presents impressive performance. Yet it lacks the ability to process extremely large graphs of which vertices can not be fit into memory. MMap [15] presents a simple approach by leveraging memory mapping mechanism in operating

systems by mapping edge and vertex data files in memory. It provides good programmability and is efficient when memory is adequate, while may suffer from drastic random writes when memory is not enough due to the random vertex access pattern. These solutions require a sorted adjacency list representation of graph, which needs time-consuming preprocessing, and SSDs to efficiently process random I/O requests. GraphQ [26] divides graphs into partitions and uses user programmable heuristics to merge partitions. It aims to answer queries by analyzing subgraphs. PathGraph [31] uses a path-centric method to model a large graph as a collection of tree-based partitions. Its compact design in storage allows efficient data access and achieves good performance on machines with sufficient memory. Galois [18] provides a machine-topology-aware scheduler, a priority scheduler and a library of scalable data structures, and uses a CSR format of graphs in its out-of-core implementation. GridGraph is inspired by these works on out-of-core graph processing, such as partitioning, locality and scheduling optimization, but is unique in its wide applicability and hardware friendliness that only limited resource is required and writes to disks are optimized. This not only provides good performance, but also protects disks from worn-out, especially for SSDs.

There are many graph engines using shared memory configurations. X-Stream [21] has its in-memory streaming engine and uses a parallel multistage shuffler to fit vertex data of each partition into cache. Ligra [23] is a shared-memory graph processing framework which provides two very simple routines for mapping over vertices and edges, inspiring GridGraph for the streaming interface. It adaptively switches between two modes based on the density of active vertex subsets when mapping over edges, and is especially efficient for applications like BFS. Polymer [32] uses graph-aware data allocation, layout and access strategy that reduces remote memory accesses and turns inevitable random remote accesses into sequential ones. While GridGraph concentrates on out-of-core graph processing, some of the techniques in these works can be integrated to improve in-memory performance further.

The 2D grid partitioning used in GridGraph is also utilized similarly in distributed graph systems and applications [10, 28, 30] to reduce communication overhead. PowerLyra [6] provides an efficient hybrid-cut graph partitioning algorithm which combines edge-cut and vertex-cut with heuristics that differentiate the computation and partitioning on high-degree and low-degree vertices. GridGraph uses grid partitioning to optimize the locality of vertex accesses when streaming edges and applies a novel 2-level hierarchical partitioning to adapt to both in-memory and out-of-core situations.

6 Conclusion

In this paper, we propose GridGraph, an out-of-core graph engine using a grid representation for large-scale graphs by partitioning vertices and edges to 1D chunks and 2D blocks respectively, which can be produced efficiently through a lightweight range-based shuffling. A second logical level partitioning is applied over this grid partitioning and is adaptive to both in-memory and out-of-core scenarios.

GridGraph uses a new streaming-apply model that streams edges sequentially and applies updates onto vertices instantly. By streaming edge blocks in a locality-friendly manner for vertices, GridGraph is able to access the vertex data in memory without involving I/O accesses. Furthermore, GridGraph could skip over unnecessary edge blocks. As a result, GridGraph achieves significantly better performance than state-of-the-art out-of-core graph systems, such as GraphChi and X-Stream, and works on both HDDs and SSDs. It is particularly interesting to observe that in some cases, GridGraph is even faster than mainstream distributed graph processing systems that require much more resources.

The performance of GridGraph is mainly restricted by I/O bandwidth. We plan to employ compression techniques[5, 14, 24] on the edge grid to further reduce the I/O bandwidth required and improve efficiency.

Acknowledgments

We sincerely thank our shepherd Haibo Chen and the anonymous reviewers for their insightful comments and suggestions. This work is partially supported by the National Grand Fundamental Research 973 Program of China (under Grant No. 2014CB340402) and the National Natural Science Foundation of China (No. 61232008).

References

[1] AVERY, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* (2011).

[2] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 44–54.

[3] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[4] BOLDI, P., SANTINI, M., AND VIGNA, S. A large time-aware web graph. In *ACM SIGIR Forum* (2008), vol. 42, ACM, pp. 33–38.

[5] BOLDI, P., AND VIGNA, S. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 595–602.

[6] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[7] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review* (1999), vol. 29, ACM, pp. 251–262.

[8] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[9] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. 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), ACM, pp. 77–85.

[10] JAIN, N., LIAO, G., AND WILLKE, T. L. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 4.

[11] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[12] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[13] KYROLA, A., BLELLOCH, G. E., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[14] LENHARDT, R., AND ALAKUIJALA, J. Gipfeli-high speed compression algorithm. In *Data Compression Conference (DCC), 2012* (2012), IEEE, pp. 109–118.

[15] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., AND KANG, U. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Big Data (Big Data), 2014 IEEE International Conference on* (2014), IEEE, pp. 159–164.

[16] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[17] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[18] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[19] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web.

[20] RANDLES, M., LAMB, D., AND TALEB-BENDIAB, A. A comparative study into distributed load balancing algorithms for cloud computing. In *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on* (2010), IEEE, pp. 551–556.

[21] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[22] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 505–516.

[23] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[24] SHUN, J., DHULIPALA, L., AND BLELLOCH, G. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Proceedings of the IEEE Data Compression Conference (DCC)* (2015).

[25] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[26] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinement. In *USENIX ATC* (2015).

[27] WANG, P., ZHANG, K., CHEN, R., CHEN, H., AND GUAN, H. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014), IEEE, pp. 562–573.

[28] XIN, R. S., GONZALEZ, J. E., FRANKLIN, M. J., AND STOICA, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[29] YAHOO. Yahoo! altavista web page hyperlink connectivity graph, circa 2002. `http://webscope.sandbox.yahoo.com/`.

[30] YOO, A., CHOW, E., HENDERSON, K., MCLENDON, W., HENDRICKSON, B., AND CATALYUREK, U. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (2005), IEEE Computer Society, p. 25.

[31] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: a path centric approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* (2014), IEEE, pp. 401–412.

[32] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[33] ZHAO, Y., YOSHIGOE, K., XIE, M., ZHOU, S., SEKER, R., AND BIAN, J. Lightgraph: Lighten communication in distributed graph-parallel processing. In *Big Data (BigData Congress), 2014 IEEE International Congress on* (2014), IEEE, pp. 717–724.

[34] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 45–58.