LUMOS: Dependency-Driven Disk-based Graph Processing

Keval Vora, *Simon Fraser University*

https://www.usenix.org/conference/atc19/presentation/vora

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

LUMOS: Dependency-Driven Disk-based Graph Processing

Keval Vora
School of Computing Science
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Abstract

Out-of-core graph processing systems are well-optimized to maintain sequential locality on disk and minimize the amount of disk I/O per iteration. Even though the sparsity in real-world graphs provides opportunities for out-of-order execution, these systems often process graphs iteration-by-iteration, hence providing Bulk Synchronous Parallel (*synchronous* for short) mode of processing which is also a preferred choice for easier programmability. Since out-of-core setting limits the view of entire graph and constrains the processing order to maintain disk locality, exploiting out-of-order execution while simultaneously providing synchronous processing guarantees is challenging. In this paper we develop a generic dependency-driven out-of-core graph processing technique, called LUMOS, that performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees. Our cross-iteration value propagation technique identifies future dependencies that can be safely satisfied, and actively computes values across those dependencies without sacrificing disk locality. This eliminates the need to load the corresponding portions of graph in future iterations, hence reducing disk I/O and accelerating the overall processing.

1 Introduction

Disk-based processing of large graphs enables processing to scale beyond the available main memory in both single machine [1, 8, 9, 13, 14, 16, 17, 24, 34, 35, 38] and cluster based [23] processing environments. With limited amount of main memory available for processing, out-of-core graph systems first divide the graph into partitions that reside on disk, and then process these partitions one-by-one by *streaming* through them, i.e., by sequentially loading them in memory and immediately processing them. As expected, such out-of-core processing is I/O intensive and systems often spend significant amount of time in loading the partitions from disk; for example, GridGraph [38], a recent state-of-art out-of-core graph processing system, spends 69-90% of time in loading edges from disk partitions.

A common concern across graph processing systems is the nature of consistency semantics they offer for programmers to correctly express their graph algorithms. Consistency semantics in the context of iterative graph processing fundamentally decide when should a vertex's value (that is computed in a given iteration) become visible to its outgoing neighbors. The most popular consistency semantics is offered by the Bulk Synchronous Parallel (BSP) [27] model (hereafter called *synchronous* processing semantics) that separates computations across iterations such that vertex values computed in a given iteration become visible to their outgoing neighbors in the next iteration, i.e., values in a given iteration are computed based on values from the previous iteration. Such clear separation between values being generated v/s values being used allows programmers to clearly reason about the important convergence and correctness properties. Hence, synchronous processing semantics often becomes a preferred choice for large-scale graph processing [18, 24, 36, 38].

While out-of-core graph processing systems that provide synchronous processing semantics have been well-optimized to maintain sequential disk locality and to minimize the amount of disk I/O per iteration, they process graphs iteration-by-iteration such that processing for a given iteration starts only after all the partitions have been processed for the corresponding previous iteration. Such synchronous processing enforces dependencies between all values across subsequent iterations. However, dependencies in graph computation are determined by the structure of the input graph, and real-world graphs are often large and sparse. This means, more often than not, two randomly chosen vertices will not be directly connected to each other, hence deeming the dependency between their values to be unnecessary in synchronous processing. This sparsity in edges provides an opportunity to perform out-of-order execution such that unrelated values across multiple iterations get simultaneously computed to amortize the disk I/O cost across multiple iterations. However, achieving such out-of-order execution in an out-of-core setting without sacrificing sequential disk locality, as well as simultaneously providing synchronous processing guarantees is challenging.

In this paper, we develop a *dependency-aware cross-iteration value propagation technique* called **LUMOS** to enable future value computations that reduce disk I/O while still guaranteeing synchronous processing semantics. We refine vertex computations into two key components: the first step performs concurrent aggregation of incoming vertex values, and the next step uses the result of concurrent aggregation to compute the final vertex value. Upon doing so, we identify that computing aggregations requires all incoming vertex values to be available which is a strong precondition that limits future computations. However, values can be safely propagated to compute *partial aggregations* that lifts off the precondition of requiring all incoming vertex values. When the partial aggregations receive all required values, they can be used to compute the final future vertex values which can be propagated further down across subsequent future iterations.

We enable such cross-iteration value propagation across partitions as partition boundaries become natural points to capture the set of value dependencies that can be safely satisfied. We further increase cross-iteration propagation via locality-aware intra-partition propagation to exploit the inherent locality in real-world graphs which has been identified in recent works [36]. While LUMOS can also correctly process asynchronous algorithms (e.g., traversal algorithms like shortest paths), we further optimize LUMOS for asynchronous algorithms by exposing relaxed processing semantics in its processing model. Finally, to achieve maximum benefits we enhance LUMOS with several key out-of-core processing strategies like selective scheduling and light-weight partitioning that have been shown to be seminal in extracting performance in out-of-core processing.

While our dependency aware cross-iteration propagation model is general enough to be incorporated in any synchronous out-of-core graph processing system, we develop LUMOS by extending GridGraph which is a state-of-art out-of-core graph processing system that guarantees synchronous processing semantics. Our evaluation shows that LUMOS is able to compute future values across 71-97% of edges which eliminates the corresponding amount of disk I/O across those iterations, and hence, LUMOS is 1.8× faster than GridGraph while it still retains the same synchronous processing semantics. To the best of our knowledge, this is the first out-of-core graph processing technique that enables future value computation across iterations, while still retaining the synchronous processing semantics throughout all the iterations, which is crucial for easy programmability.

2 Background & Motivation

We first discuss about semantics of synchronous execution, and then summarize out-of-core graph processing techniques.

2.1 Synchronous Processing Semantics

The Bulk Synchronous Parallel (BSP) model [27] is a popular processing model that provides synchronous stepwise execution semantics with separated computation and communication/synchronization phases. Under the BSP model, values in a given iteration are computed based on values from the previous iteration. We illustrate the *synchronous processing semantics* of BSP model using the PageRank algorithm as an example [1] in Algorithm 1. The algorithm computes vertex values (`newPr`) using the ones computed in previous iteration (`pr`) as shown on line 6. The flow of values across iterations is explicitly controlled via SWAP() on line 11.

Algorithm 1 Synchronous PageRank

```
1:  G = (V, E)                                   ▷ Input graph
2:  pr = {1, 1, ..., 1}                          ▷ Floating-point array of size |V|
3:  while not converged do
4:      newPr = {0, 0, ..., 0}                   ▷ Floating-point array of size |V|
5:      par-for (u, v) ∈ E do
6:          AtomicAdd(&newPr[v], pr[u] / |out_neighbors(u)|)
7:      end par-for
8:      par-for v ∈ V do
9:          newPr[v] = 0.15 + 0.85 × newPr[v]
10:     end par-for
11:     Swap(pr, newPr)
12: end while
```

Such clear separation of values being generated v/s values being used allows programmers to clearly reason about the important convergence and correctness properties. Hence, synchronous processing semantics often becomes a preferred choice for large-scale graph processing [18, 24, 36, 38].

2.2 Out-of-Core Graph Processing

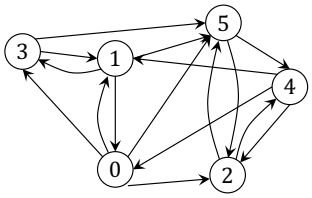
Disk-based graph processing has been a challenging task due to ever growing graph sizes. The key components in efficient out-of-core graph processing systems is a disk-friendly partition-based data-structure, and an execution engine that processes the graph in a partition-by-partition fashion that maximizes sequential locality. Figure 1 shows how a given graph is represented as partitions on disk. Each partition represents incoming edges for a range of vertices (*chunk*-based partitioning [38]); partition p_0 holds incoming edges for vertices 0 and 1, p_1 holds for vertices 2 and 3, and p_3 for vertices 4 and 5. The iterative engine processes the graph by going through these partitions in a fixed order; it sequentially loads edges from partition p_0 to process them in memory, then from partition p_1, and finally from the last partition p_2. Once all the partitions are processed, the iteration ends by performing computations across vertex values, that may reside on disk or in-memory (depending on availability of memory). This entire process is repeated for multiple iterations until algorithm-specific termination condition is satisfied.

Several works aim to improve out-of-core graph processing [1, 8, 9, 13, 14, 16, 17, 24, 31, 32, 34, 35, 38] as summarized in Table 1. Depending on the processing semantics they offer, they fall in two categories:

[1] Algorithm 1 is simplified to eliminate details like selective scheduling.

	Out-of-Order Execution	Future Value Computation	Synchronous Semantics	Async. Algorithms (e.g., SSSP)
GraphChi [13]	✗	✗	✓	✓
X-Stream [24]	✗	✗	✓	✓
GridGraph [38]	✗	✗	✓	✓
FlashGraph [35]	✗	✗	✓	✓
TurboGraph [8]	✗	✗	✓	✓
Mosaic [17]	✗	✗	✓	✓
GraFBoost [9]	✗	✗	✓	✓
Graphene [15]	✗	✗	✓	✓
Garaph [16]	✗	✗	✓	✓
DynamicShards [31]	✓	✗	✓	✓
Wonderland [34]	✓	✓	✗	✓
CLIP [1]	✓	✓	✗	✓
AsyncStripe [4]	✓	✓	✗	✓
LUMOS	✓	✓	✓	✓

Table 1: Key characteristics of existing out-of-core graph processing systems and LUMOS.



(a) Example graph.

p_0	p_1	p_2
(0, 1)	(0, 3)	(0, 5)
(1, 0)	(0, 2)	(1, 5)
(3, 1)	(1, 3)	(2, 4)
(4, 0)	(4, 2)	(2, 5)
(4, 1)	(5, 2)	(3, 5)
		(5, 4)

(b) Edge partitions.

Figure 1: An example graph partitioned into three partitions (p_0, p_1 and p_2) that reside on disk.

(A) Synchronous Out-of-Core Graph Processing.
GraphChi [13], X-Stream [24], GridGraph [38], and others provide synchronous processing semantics. While initial systems like GraphChi [13] and X-Stream [24] proposed efficient processing models, their performance is limited due to management of edge-scale intermediate values in every iteration (edge values in GraphChi and edge updates in X-Stream).

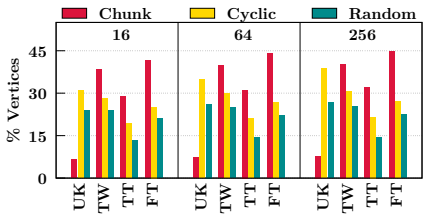
GridGraph [38] is an efficient out-of-core graph processing system that eliminates edge-scale intermediate updates. It divides vertices into subsets called *chunks* and partitions edges into 2D grid based on these chunks. In Figure 1b, the dashed-horizontal lines represent boundaries of *blocks* such that the entire representation becomes a 2D grid. The 2D grid is processed by streaming through edge-blocks. Furthermore, GridGraph enables selective scheduling which eliminates unnecessary edges to be loaded from disk by skipping partitions. Since its processing model is designed to minimize disk I/O, it is the state-of-art out-of-core graph processing system that provides synchronous processing semantics.

(B) Asynchronous Out-of-Core Graph Processing.
Recent works like CLIP [1] and Wonderland [34] are customized for asynchronous algorithms like path-based algorithms (e.g., BFS and shortest paths). These frameworks leverage the algorithmic properties (e.g., monotonicity [28, 29]) to process partitions based on newly computed values, resulting in faster convergence. CLIP [1] processes partitions multiple times in memory while Wonderland [34] performs abstraction-guided processing for faster information propagation. Even though these techniques perform out-of-order computations (see Table 1), they do not provide synchronous processing semantics since they violate the processing order across computed vertex values. Hence, they cannot be used for synchronous graph algorithms.

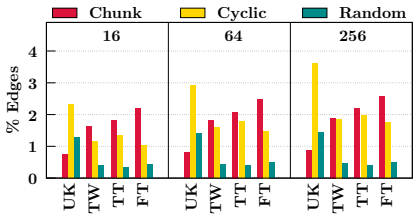
Limitations with Out-of-Core Systems.
As shown in Table 1, none of the systems perform *future value computation* (i.e., beyond a single iteration) while simultaneously providing synchronous processing semantics. In synchronous out-of-core frameworks, the processing model is tied down to strict iteration-by-iteration processing. Such tight coupling between synchronous semantics and strict processing order limits the performance of out-of-core graph processing. Particularly, the sparsity in real-world graphs often presents opportunities to proactively compute future values based on when value dependencies get resolved; realizing such processing across future iterations can be beneficial in out-of-core setting since edges corresponding to values that have already been computed for future iterations do not need to be loaded in those iterations, hence directly reducing disk I/O. However, such acceleration via future value computation is not achieved in out-of-core systems due to their strict iteration-by-iteration processing.

It is crucial to note that out-of-order execution does not necessarily result in computing across future values. Specifically, DynamicShards [31] performs out-of-order execution to dynamically capture the set of active edges to be processed in a given iteration. It achieves this by dropping inactive (or useless) edges across iterations and delaying computations that cannot be performed due to missing edges. While the delayed computations get periodically processed in shadow iterations (i.e., out-of-order execution), they do not compute across future values to leverage sparse dependencies. Asynchronous systems [1,34], on the other hand, do compute beyond a single iteration, but do not provide processing semantics.

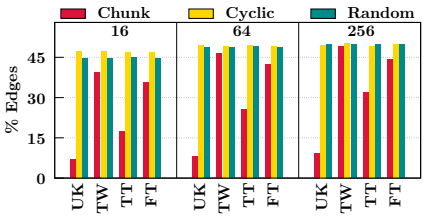
This poses an important question: *how to process beyond a single iteration to reduce disk I/O in out-of-core processing while simultaneously guaranteeing synchronous processing semantics?*



(a) Vertices for which v^{t+1} can be computed in iteration t using $u^t, \forall (u,v) \in E$.



(b) Incoming edges for vertices in Figure 2a that don't need to be loaded from disk in iteration $t+1$.



(c) Edges that don't need to be loaded from disk in iteration $t+1$, when computing partial aggregation in iteration t as shown in Eq. 2.

Figure 2: Percentage of vertex computations and edge savings across three light-weight partitioning strategies (chunking, cyclic hashing and random partitioning) and three partition sizes (#partitions = 16, 64 and 256) on four large graphs (UK, TW, TT and FT from Table 4).

3 LUMOS: Dependency-Driven Cross-Iteration Value Propagation

Since our goal is to provide synchronous processing semantics while overlapping computations across multiple iterations, we first characterize cross-iteration dependencies to capture synchronous semantics and then develop our out-of-core value propagation strategy that guarantees those captured semantics.

3.1 Characterizing Synchronous Dependencies

Synchronous iterative graph algorithms compute vertex values in a given iteration based on values of their incoming neighbors that were computed in the previous iteration. Since computations are primarily based on graph structure, such cross iteration dependencies can be captured via the graph structure as follows:

$$\forall (u,v) \in E, \quad u^t \mapsto v^{t+1} \tag{1}$$

where u^t and v^{t+1} represent values of vertex u in iteration t and vertex v in iteration $t+1$ respectively, and $\mapsto$ indicates that v^{t+1} is value-dependent on u^t. It is important to note that there are no dependency relationships among vertices that are not directly connected by an edge. With limited view of graph structure available at any given time in out-of-core graph processing, the cross-iteration dependencies get satisfied by processing vertices and edges in a given iteration only after processing for the previous iteration is completed.

3.2 Out-of-core Value Propagation

We ask two important questions that allow us to identify the interplay between cross-iteration value propagations to satisfy future dependencies and partition-by-partition processing orchestrated by out-of-core graph systems.

3.2.1 When to propagate?

A straightforward way to enable processing beyond a given single iteration t is to compute vertex values for the subsequent iteration $t+1$ if incoming neighbors' values corresponding to t are available at the time when those vertices are processed. With partition-by-partition out-of-core processing, we know that values for vertices belonging to a given partition p become available when p is processed; hence, we can allow outgoing neighbors to use these available values and compute for subsequent iterations if their partitions get processed after p in the same iteration. While theoretically this appears to be a promising direction, we profile the large graphs from Table 4 to measure the number of vertices for which future values can be computed (shown in Figure 2a), and the number of edges that don't need to be loaded for the corresponding vertices in future iterations (shown in Figure 2b). To eliminate the impact of partitioning, we profiled across three light-weight partitioning schemes, chunk-based partitioning (as used in [38]), cyclic partitioning (where vertex ids are hashed to partitions) and random partitioning; and, across three partition sizes corresponding to number of partitions being 16, 64 and 256. Even though Figure 2a shows up to 45% of vertices can compute values for subsequent iterations, it contributes to only 1-4% of edge savings as shown in Figure 2b. This means, future values can be computed for vertices that have low in-degree and, as expected, high in-degree vertices cannot compute future values since values for all of their incoming neighbors do not become available in time.

To achieve high amount of cross-iteration value propagation, we want to relax the precondition such that availability of *all* incoming neighbor values does not become a requirement. We achieve this by computing only the aggregated values for future iterations instead of computing final vertex values. Let $\bigoplus$ denote the aggregation operator that computes the intermediate value based on incoming neighbors and f denote the function to compute vertex's value based on aggregated intermediate value. For example, in PageRank (Algorithm 1), ATOMICADD on line 6 represents $\bigoplus$ and line 9 shows f. In a given iteration t, we aim to compute [2]:

$$v^t = f(\bigoplus_{\forall e=(u,v)\in E} (u^{t-1})) \quad \text{and} \quad g(v^{t+1}) = \bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u)<p(v)}} (u^t) \tag{2}$$

[2] Values residing on edges (i.e., edge weights) have been left out from equations for simplicity as they do not impact cross-iteration dependencies.

where $g(v)$ represents aggregated value of v and $p(v)$ is the partition to which v belongs. It is important to note that $\bigoplus_{\forall e=(u,v)\in E}$ represents a complete aggregation while $\bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u)<p(v)}}$ represents partial aggregation as the precondition $p(u) < p(v)$ may not be satisfied by all edges. Since $\forall u \in V, u^{t-1}$ is available in iteration t (due to barrier semantics), we can perform complete aggregation and also compute the vertex's value using f. However, since u^t becomes available as partitions get processed in the same iteration t, at a given point in time only the available u^t values can be propagated to compute the partial aggregation which satisfies the future cross-iteration dependency $u^t \mapsto v^{t+1}$. Since typically partitions get processed in numerical order, with P being total number of partitions, we know that $\forall i, j \in [0, P)$, if $i < j$, partition i gets processed before partition j. This ordering is captured in the precondition for partial aggregation $g(v^{t+1})$. Hence, in Figure 1, as p_0, p_1 and p_2 get processed in that order, v_0^t is available for $g(v_2^{t+1})$, $g(v_3^{t+1})$ and $g(v_5^{t+1})$ via $(0,2)$, $(0,3)$ and $(0,5)$ respectively, while v_3^t is not available for $g(v_1^{t+1})$ during iteration t.

As shown in Figure 2c, the percentage of values propagated (via partial aggregation) increases to 40-50%; the edges corresponding to these propagations need not be loaded in the subsequent iteration (i.e., directly reducing disk I/O), which is significantly higher compared to that in Figure 2b. We also observe that random partitioning compares well with other techniques and enables higher cross-iteration propagation; this can be reasoned with the high chances of an edge (u,v) being placed across partitions such that $p(u) < p(v)$. In Section 5.3, we will explore more light-weight partitioning strategies that will further enable cross-iteration propagation.

Note that the above value propagation requires vertex values for the current iteration to be computed as partitions get processed. We developed our processing model to simultaneously compute vertex values as corresponding partition's edges get processed (discussed further in Section 5.1).

3.2.2 How far (in iteration space) to propagate?

Cross-iteration dependencies are linear in iteration space and hence, we can potentially propagate values for future iterations beyond $t+1$. In order to guarantee synchronous processing semantics, we need to ensure that v^{t+1} gets computed in iteration t before it is further propagated to out-neighbors of v. We define a value v^x to be *computable* when all its incoming values corresponding to iteration $x-1$ have been propagated to $g(v^x)$ (i.e., a complete aggregation has occurred in $g(v^x)$). Since out-of-core processing propagates values to vertices based on its partitions, values can become computable when the corresponding vertex's partition gets processed. Computable values get computed by applying f on $g(v^x)$ to achieve v^x which can be further propagated to out-neighbors of v for $x+1$. For example in Figure 1, vertex 3 has two incoming edges $(0,3)$ and $(1,3)$, both of which contribute to $g(v_3^{t+1})$ during iteration t; hence, we can compute $v_3^{t+1} = f(g(v_3^{t+1}))$ during iteration t itself and further propagate v_3^{t+1} across the outgoing edge $(3,5)$ in the same iteration t.

For a given iteration t, we know that v^t becomes computable when $p(v)$ gets processed. Hence, for any arbitrary k, v^{t+k} becomes computable when $\forall (u,v) \in E, p(u) < p(v)$ and u^{t+k-1} is computable. This is because the partial aggregation $\bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u)<p(v)}}$ becomes equivalent to complete aggregation $\bigoplus_{\forall e=(u,v)\in E}$ when $\forall (u,v) \in E, p(u) < p(v)$. These computable values that get further propagated lead to I/O reduction in the corresponding future iterations. We define distance of propagation (in iteration space) based on the difference between the current iteration number and the iteration number for which any value propagation occurs in the current iteration. Formally, the *distance of propagation D* is defined as:

$$D = \max_{\forall t} \left(\max_{\forall g(v^{t+k})} (k+1) \right)$$

In traditional out-of-core graph systems, barriers across iterations ensure that $D = 1$. Our dependency-aware cross-iteration propagation achieves $D > 1$; for example, with propagation for immediately subsequent iteration, D is 2. Table 2 summarizes the percentage of edge propagations that occur across distances 2, 3 and 4. As we can see, propagations decrease drastically at distance 3, and are close to 0% after that. Since achievable benefits are minor beyond propagation distance 2, we perform cross-iteration propagation until distance 2, i.e., for the current iteration t and the next iteration $t+1$.

	UK	TW	TT	FT
$D = 2$	6.8 - 49.7	39.2 - 50.1	17.4 - 49.7	35.6 - 49.8
$D = 3$	0.38 - 0.52	0.13 - 0.25	0.11 - 0.44	0.04 - 0.26
$D = 4$	< 0.01	< 0.01	< 0.01	< 0.01

Table 2: Percentage (min-max range) of edge propagations across three partitioning strategies from Figure 2.

3.3 Graph Layout

Since cross iteration dependencies are primarily based on the input graph structure, value propagations beyond current iteration can be statically determined based on the precondition involved in Eq. 2. In order to completely avoid reading edges whose dependencies have been satisfied, we create separate graph layouts for subsequent iterations. In this way, the execution switches between different graph layouts on disk. In theory, we can create D separate graph layouts to propagate values across D iterations; however, to simplify exposition we discuss the layout for $D = 2$ since larger propagation distances provide diminishing benefits (as discussed in Section 3.2). With $D = 2$, we have two graph layouts: the *primary* layout consisting of all edges, and the *secondary* layout containing only subset of edges. Figure 3 shows the secondary layout and its corresponding graph representation for the primary graph

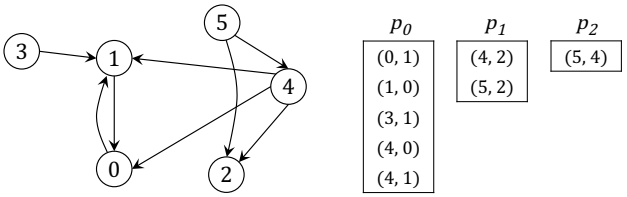


Figure 3: Secondary layout for graph from Figure 1.

Graph	Without Intra-Partition Propagation	With Intra-Partition Propagation
UK	6.84 - 49.69	49.76 - 92.47
TW	39.24 - 50.09	50.31 - 62.22
TT	17.43 - 49.74	43.32 - 61.87
FT	35.58 - 49.80	47.20 - 56.19

Table 3: Percentage (min-max range) of edge propagations with and without intra-partition propagation across three partitioning strategies from Figure 2.

layout in Figure 1. The secondary layout does not contain 8 out of 16 edges since the corresponding dependencies get satisfied while processing primary layout.

We analyze the reduction in I/O caused by our secondary graph layout. Since $|E| >> |V|$, out-of-core graph processing systems like GridGraph achieve high performance by ensuring that the edges are loaded only once in every iteration. Hence, for each iteration, their I/O amount is $|E| + k \times |V|$ where $k \geq 2$ captures batch loading of vertices for each partition. While our I/O amount for the iteration using primary layout remains the same, it reduces significantly when secondary layout is used. Let α be the ratio of edges for which values are propagated for subsequent iteration when primary layout is processed. Since those edges are not present in secondary layout, the I/O amount directly reduces by $\alpha \times |E|$. Hence, the total I/O amount is:

$$C = \begin{cases} (1-\alpha) \times |E| \ + \ k \times |V| & \ldots \text{ secondary layout} \\ |E| \ + \ k \times |V| & \ldots \text{ primary layout} \end{cases}$$

As we will see in Section 6, α is typically over 0.7 which drastically reduces the I/O amount.

3.4 Intra-Partition Propagation

So far cross-iteration propagation is performed for edges across partitions as dictated by Eq. 2. While this captures a large subset of edges that don't need to be loaded in the subsequent iteration, it has been recently shown in [36] that real-world graphs often possess natural locality such that adjacent vertices are likely to be numbered close to each other due to the traversal nature of graph collection strategies (e.g., crawling, anonymization, etc.). This means, chunk-based partitioning strategies where contiguous vertex-id ranges get assigned to partitions have several edges such that both endpoints belong to the same partition.

In order to leverage this natural locality, in a given iteration t, we aim to propagate value for subsequent iteration across edge (u,v) where $p(u) = p(v)$. However, since u^t becomes available as partition $p(u)$ gets processed in the same iteration t, u^t can only be propagated after $p(u)$ has been processed. Hence, if (u,v) can be held in memory until $p(u)$ gets fully processed, we can propagate u^t to satisfy the future cross-iteration dependency $u^t \mapsto v^{t+1}$. This means, intra-partition cross-iteration propagation relaxes our precondition for partial aggregation to become:

$$g(v^{t+1}) = \bigoplus_{\substack{\forall e=(u,v) \in E \\ s.t.\ p(u) \leq p(v)}} (u^t) \qquad (3)$$

Table 3 summarizes the increase in cross-iteration propagation when intra-partition propagation is enabled. As we can see, cross-iteration propagation increases to 43-92% with intra-partition propagation, which further saves disk I/O.

We enable intra-partition cross-iteration value propagation by ensuring that partition sizes remain small enough such that they can be entirely held in memory. As shown in Section 5.1, our processing model holds the partition in memory until it gets processed, and then performs cross-iteration propagation for edges whose both end-points belong in the partition.

3.5 Value Propagation v/s Partition Size

The amount of cross-iteration value propagation not only depends on the partitioning strategy, but also varies based on the size of partitions. As we can observe in Figure 2c, the same partitioning strategy with different partition sizes enables different amount of cross-partition propagation. In general, since cross-partition propagation is achieved on edges crossing partition boundaries, smaller partitions lead to higher cross-partition propagation compared to that by larger partitions. As an extreme case, each vertex residing in its own partition would result in smallest partitions, and would cause highest amount of cross-partition value propagation.

Intra-partition propagations, on the other hand, are amenable to large partitions as they enable propagations across edges within the partitions. Even though smaller partitions allow lesser intra-partition propagations compared to that allowed by larger partitions, the set of edges crossing smaller partitions is larger, hence allowing cross-partition propagations through edges that get missed by intra-partition propagations. It is interesting to note that as partitions become smaller, edges that were originally participating in intra-partition propagation get cut across partition boundaries either in forward direction (i.e., $p(u) < p(v)$ for edge (u,v)) or in backward direction. Since cross-partition value propagation occurs on forward edges only, the overall cross-iteration value propagation (combined cross-partition and intra-partition) reduces as partitions become smaller. Hence, it is preferable to have large partitions that can fit in main memory to maximize cross-iteration value propagation. In Section 5.3, we propose light-weight partitioning heuristics that increase the amount of cross-iteration value propagation, which in turn reduces the severity of partition size's impact.

3.6 LUMOS with DynamicShards

LUMOS can be combined with DynamicShards [31] to perform cross-iteration value propagation with dynamic edge selection. Specifically, primary layout partitions can be made dynamic (by dropping inactive edges) while secondary layout partitions can be left static since they are already small in size. Furthermore, shadow iterations require the missing graph edges to process delayed computations, and hence, shadow iterations must be scheduled during primary iterations in order to amortize loading costs. It is interesting to note that even though secondary layouts are kept static, computations from secondary layouts can get delayed due to transitive delays occurring from previous iterations' primary layouts.

4 LUMOS for Asynchronous Algorithms

Several graph analytics algorithms like traversal based monotonic algorithms (e.g., shortest paths, BFS, connected components, etc.) are asynchronous, i.e., they do not require synchronous processing semantics to guarantee correctness [28, 30]. Nevertheless, having synchronous semantics does not hurt the correctness guarantees for asynchronous algorithms because synchronous execution is one of the legal executions under the set of asynchronous executions. In other words, the dependencies enforced by synchronous processing semantics in Eq. 1 are only stricter (while still being legal) than that required by asynchronous semantics. Since LUMOS's out-of-order value propagation technique does not violate the dependencies defined in Eq. 1, LUMOS works correctly on asynchronous algorithms like shortest paths and connected components as well.

Furthermore, LUMOS can be optimized to efficiently process asynchronous algorithms by incorporating relaxed processing semantics in its processing model. For example, traversal based monotonic algorithms often rely on selection functions [29] like MIN() that enable vertex's value to be computed based on value coming from its single incoming edge. This means, intermediate vertex values computed based on subset of their incoming values represent a valid final value (instead of a partial aggregation) that can be instantly propagated to outgoing edges. In other words, computations in Eq. 2 do not require separation of values across iterations t and $t+1$, and $g(v^t)$ can be directly incorporated in v^t. Hence, LUMOS can enable value propagation for asynchronous algorithms by maintaining a single version of vertex values and directly propagating the updated values across edges.

Such *asynchronous value propagation* is achieved both across (inter-partition) and within (intra-partition) partitions; intra-partition propagations are achieved by recomputing over in-memory partition as described in Section 3.4. An interesting side effect of asynchronous value propagation is that secondary graph layout is no longer necessary because any value propagation across edges in the primary graph layout does not violate asynchronous semantics, and hence, entire processing can occur on the primary graph layout itself.

5 The LUMOS System

So far we discussed value propagation in a generalized out-of-core processing context without focusing on any particular system. This makes our proposed cross-iteration value propagation techniques useful for existing out-of-core processing systems. We now discuss the important design details involved in incorporating LUMOS into GridGraph. We choose GridGraph since its streaming-apply processing model is designed to minimize I/O amount, making it state-of-art synchronous out-of-core graph processing system.

5.1 Propagation based Processing Model

LUMOS's processing model is similar to out-of-core processing systems where edges are loaded from disk in batches and processed in memory. We discuss the processing model at partition level to showcase how cross-iteration propagations are performed. Algorithm 2 shows how primary and secondary partitions are processed; LUMOS offers three key programming interfaces: (a) PROCESSPRIMARY performs standard propagation and cross-iteration propagation across primary partitions; (b) PROCESSSECONDARY performs standard propagations across secondary partitions; and, (c) VERTEXMAP performs updates across vertices. PROCESSPRIMARY processes both, edges (lines 4, 6, 13) and vertices (line 9). PROCESSSECONDARY's structure is kept similar to PROCESSPRIMARY to enable easier programmability of graph algorithms. The primary partitions get processed in even iterations using PROCESSPRIMARY while the secondary partitions get processed in odd iterations using PROCESSSECONDARY. Figure 4 illustrates how partition-by-partition processing is achieved using primary and secondary layouts across two consecutive iterations. Along with the traditional cross-iteration barriers, we also have cross-partition barriers while processing primary partitions to ensure that precondition in Eq. 2 gets correctly satisfied for cross-iteration propagation; these cross-partition barriers are not required while processing secondary partitions. Algorithm 3 shows PageRank algorithm using our propagation interface. Beyond the standard edge function PROPAGATE and vertex function COMPUTE, we also use the cross-iteration edge propagation function CROSSPROPAGATE. While the shape of CROSSPROPAGATE is similar to PROPAGATE, they operate on different values for the same edge, i.e., PROPAGATE aggregates for current iteration while CROSSPROPAGATE aggregates for subsequent iteration. The aggregation for subsequent iteration is made available using ADVANCE (line 13).

5.2 Selective Scheduling

One of the strengths of 2D grid layout is that it enables selective scheduling of edge computations so that edge-blocks can be skipped to reduce unnecessary I/O [38]. LUMOS carefully incorporates selective scheduling with cross-iteration propagation to ensure that scheduling gets correctly managed for primary and secondary partitions.

Algorithm 2 Propagation Interface

```
1:  function processPrimary(propagate,
                            crossPropagate, compute)
2:    for partition ∈ primaryPartitions do
3:      par-for edge ∈ partition do
4:        propagate(edge)
5:        if p(edge.source) < p(edge.target) then
6:          crossPropagate(edge)
7:        end if
8:      end par-for
9:      vertexMap(compute, vertex_chunk(partition))
10:     if Locality-Aware Intra-Partition Propagation then
          /* partition is held in memory (see Section 3.4) */
11:       par-for edge ∈ partition do
12:         if p(edge.source) = p(edge.target) then
13:           crossPropagate(edge)
14:         end if
15:       end par-for
16:     end if
17:   end for
18: end function
19: function processSecondary(propagate, compute)
20:   for partition ∈ secondaryPartitions do
21:     par-for edge ∈ partition do
22:       propagate(edge)
23:     end par-for
24:     vertexMap(compute, vertex_chunk(partition))
25:   end for
26: end function
27: function vertexMap(vFunc, VS = V)        ▷ V is default arg.
28:   sum = 0
29:   par-for v ∈ VS do
30:     sum += vFunc(v)
31:   end par-for
32:   return sum
33: end function
```

Algorithm 3 PageRank Example

```
1:  function propagate(e)
2:    atomicAdd(&sum[e.target], pagerank[e.source] / outdegree[e.source])
3:  end function
4:  function crossPropagate(e)
5:    atomicAdd(&secondary_sum[e.target],
                sum[e.source] / outdegree[e.source])
6:  end function
7:  function compute(v)
8:    sum[v] = 0.15 + 0.85 × sum[v]
9:  end function
10: function advance(v)
11:   diff = |pagerank[v] − sum[v]|
12:   pagerank[v] = sum[v]
13:   sum[v] = secondary_sum[v]
14:   secondary_sum[v] = 0
15:   return diff
16: end function
17: pagerank = [1, ..., 1]
18: sum = [0, ..., 0]
19: secondary_sum = [0, ..., 0]
20: iteration = 0
21: converged = false
22: while ¬converged do
23:   if iteration % 2 == 0 then
24:     processPrimary(propagate, crossPropagate,
                       compute);
25:   else
26:     processSecondary(propagate, compute);
27:   end if
28:   d = vertexMap(advance);
29:   converged = d / |V| ≤ threshold
30:   iteration = iteration + 1
31: end while
```

An *active* edge-block represents edges that will be loaded from disk; otherwise, they will be skipped. When processing primary partitions (i.e., during even iterations), depending on the state (active/inactive) of an edge-block for primary and secondary layouts at the time when it needs to be processed, there can be four cases. While three of those cases can be handled in the same way as done for primary partitions, the case when the edge-block is inactive for primary layout but is active for secondary layout need to be considered carefully. In this case, while processing secondary partitions in the subsequent iteration, the corresponding edge-block gets loaded from the primary layout instead of secondary layout to ensure that all necessary edges within the edge-block participate correctly in value propagation. LUMOS maintains this cross-iteration selective scheduling information using 2-bits per edge-block; first bit indicating whether the edge-block is active or inactive, and second bit indicating whether to load edge-block from primary layout or secondary layout.

5.3 Graph Layout & Partitioning

For a graph $G = (V, E)$, V is divided into P disjoint subsets of vertices (called *chunks*), $C = \{c_0, c_1, ..., c_{P-1}\}$ such that $\bigcup_{c \in C} c = V$ and $\forall c_i, c_j \in C, c_i \cap c_j = \varnothing$. The edges are represented as a 2D grid of $P \times P$ edge-blocks on disk. An edge (u, v) is in edge-block b_{ij} if $u \in c_i \wedge v \in c_j$. It is important to note that a column i in the 2D grid has incoming edges for vertices belonging to c_i, and similarly a row j has outgoing edges for vertices belonging to c_j. Hence, each column is a partition for LUMOS in accordance with the precondition in Eq. 2 that satisfies the dependency relation. Similarly, each row is a partition for LUMOS when values need to be propagated across incoming edges (i.e., a transposed view).

As discussed in Section 3.3, we create a primary layout and a secondary layout. This means, we create two separate 2D grids on disk, one for each layout. An important issue in creating grid layouts is partitioning V into P chunks. Out-of-core processing systems use a simplified structure-oblivious partitioning strategy based on vertex-id ranges, i.e., assuming vertices are numbered between 0 to $|V|$, chunks are formed as contiguous range of vertex numbers: vertices 0 to $k-1$ form chunk 0, vertices k to $2k-1$ form chunk 1, and so on. While such structure-oblivious partitioning enables good amount of cross-iteration propagation (shown in Section 3), we develop greedy partitioning strategies that carefully use the vertex degree information to maximize cross-iteration value propagation.

Let $\mathscr{P} = \{p(v_0), p(v_1), ..., p(v_{|V|-1})\}$ capture the partition-

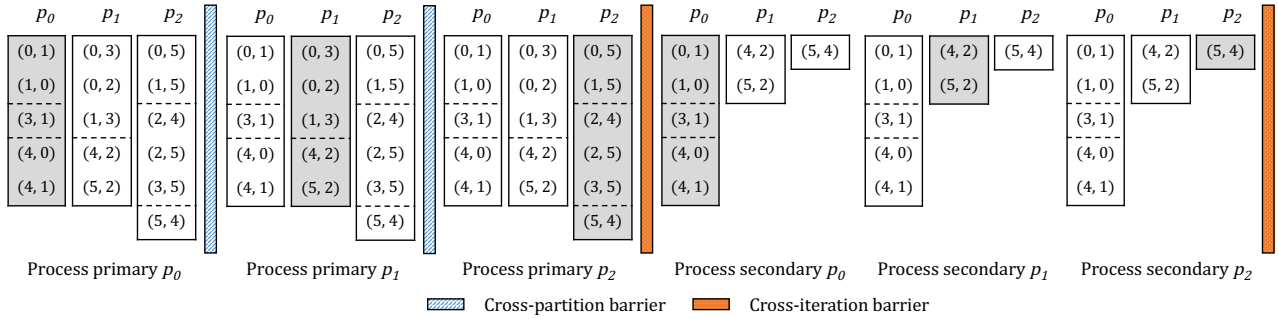


Figure 4: Execution using primary and secondary layouts across two consecutive iterations.

ing information that maps vertices in V to chunks in C. We define our partitioning objective as:

$$\arg\max_{\mathscr{P}} |\{(u,v) : (u,v) \in E \ \wedge \ p(u), p(v) \in \mathscr{P} \ \wedge \ p(u) < p(v)\}| \quad (4)$$

Note that with intra-partition propagation, the above condition becomes $p(u) \leq p(v)$ and there is an additional constraint to limit partition sizes:

$$\forall c_i \in C, \sum_{\substack{\forall v \in c_i \ \wedge \\ \forall (u,v) \in E}} (u,v) \ < \ T$$

where T is a threshold based on available memory.

Since the condition in Eq. 4 can be directly viewed as u's outgoing edges contributing to cross-iteration dependencies, a straightforward greedy heuristic can be to place vertices with higher out-degree in earlier partitions. However, an interesting dual to this reasoning can be that v's incoming edges contribute to cross-iteration dependencies and hence, the greedy heuristic can be to place vertices with higher in-degree in later partitions. Based on these insights, we develop three key partitioning heuristics to assign vertices in V to chunks in C:

(A) Highest Out-Degree First: $\forall u, v \in V$,
if $p(u) < p(v)$, then $out\_degree(u) \geq out\_degree(v)$

(B) Highest In-Degree Last: $\forall u, v \in V$,
if $p(u) < p(v)$, then $in\_degree(v) \geq in\_degree(u)$

(C) Highest Out-Deg. to In-Deg. Ratio First: $\forall u, v \in V$,
if $p(u) < p(v)$, then $\frac{out\_degree(u)}{in\_degree(u)} \geq \frac{out\_degree(v)}{in\_degree(v)}$

Note that the above heuristics are simpler than structure-based partitioning since computing degrees of vertices only requires a single pass over the edge list.

6 Evaluation

We evaluate LUMOS using billion-scale graphs and synchronous graph algorithms, and compare its performance with GridGraph [38] which is the state-of-art out-of-core graph processing framework that provides synchronous semantics.

Graphs	Disk Size	|E|	|V|
UKDomain (UK) [2]	6.9-20.7GB	1.0B	39.5M
Twitter (TW) [12]	11-33GB	1.5B	41.7M
TwitterMPI (TT) [3]	15-45GB	2.0B	52.6M
Friendster (FT) [5]	20-60GB	2.5B	68.3M
Yahoo (YH) [33]	50-150GB	6.6B	1.4B
RMAT29 (RM)	66-198GB	8.6B	537M

Table 4: Real world & synthetic graphs.

6.1 Experimental Setup

LUMOS's evaluation is carried out across three different AWS EC2 storage optimized instances. For performance experiments (Section 6.2), we use h1.2xlarge with 8 vCPUs, 32GB memory and 2TB HDD. Its disk sequential read bandwidth [3] is 278MB/sec whereas the memory subsystem read bandwidth is 9.6GB/sec. To study the effect of I/O scaling (Section 6.3), we use d2.4xlarge and i3.8xlarge instances. The d2.4xlarge instance is used with 16 vCPUs, 32GB memory and 1 to 4 2TB HDDs providing disk bandwidth of 195MB/sec to 768MB/sec, whereas the i3.8xlarge instance is used with 32 vCPUs, 64GB memory and 1 to 4 1.9TB SSDs providing disk bandwidth of 1.2GB/sec to 3.9GB/sec.

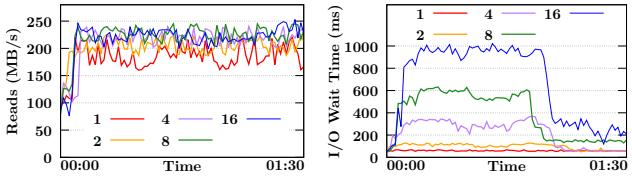
We use six synchronous graph algorithms. PageRank (PR) [22] is an algorithm to rank web-pages while Weighted Pagerank (WPR) is its variant where edges have weights as applied for social network analysis. Co-Training Expectation Maximization (CoEM) [21] is a semi-supervised learning algorithm for named entity recognition. Belief Propagation (BP) [10] is an inference algorithm to determine states of vertices based on sum of products. Label Propagation (LP) [37] is a learning algorithm while Dispersion (DP) [11] is a simulation based information dispersion model. We run each algorithm for 10 iterations; PR and DP operate on unweighted graphs while CoEM, LP, WPR and BP require weighted graphs. This adds 4 bytes per edge for CoEM, LP and WPR, and 16 bytes per edge for BP; hence, increasing graph sizes to 1.5× and 3× respectively.

We evaluate LUMOS using billion scale graphs from Ta-

[3] Disk sequential read bandwidth measured using `hdparm`.

	Version	TT	FT	YH
PR	**GridGraph**	737	1008	3223
	LUMOS-BASE	563	659	2027
	LUMOS	439	583	1885
	× **LUMOS**	**1.68**×	**1.73**×	**1.71**×
CoEM	**GridGraph**	1119	1554	5082
	LUMOS-BASE	861	1029	3216
	LUMOS	651	914	3043
	× **LUMOS**	**1.72**×	**1.70**×	**1.67**×
DP	**GridGraph**	846	1032	3484
	LUMOS-BASE	656	675	2219
	LUMOS	498	611	2111
	× **LUMOS**	**1.70**×	**1.69**×	**1.65**×
BP	**GridGraph**	2498	3782	13769
	LUMOS-BASE	1921	2456	8660
	LUMOS	1487	2212	7913
	× **LUMOS**	**1.68**×	**1.71**×	**1.74**×
WPR	**GridGraph**	984	1302	4330
	LUMOS-BASE	769	874	2758
	LUMOS	569	770	2547
	× **LUMOS**	**1.73**×	**1.69**×	**1.70**×
LP	**GridGraph**	1054	1421	4583
	LUMOS-BASE	805	935	2976
	LUMOS	624	826	2728
	× **LUMOS**	**1.69**×	**1.72**×	**1.68**×

Table 5: Execution times (in seconds) for LUMOS, LUMOS-BASE and GridGraph. Bold numbers indicate speedups of LUMOS over GridGraph.



(a) Throughput. (b) I/O Wait.

Figure 5: I/O throughput and waiting times for PR on TT across 1, 2, 4, 8 and 16 threads.

ble 4. For experiments on h1.2xlarge, we use TT, FT and YH whereas RMAT29 is used for I/O scaling experiments. Since unweighted TT fits within 32 GB (on h1.2xlarge), we limit the memory of unweighted TT experiments to 16 GB.

We compare the following three versions:

1. **GridGraph (GG):** is the GridGraph system [38].
2. **LUMOS-BASE (LB):** is LUMOS based on Eq. 2. It performs cross-iteration propagation only across partitions.
3. **LUMOS (L):** is LUMOS based on Eq. 3. It also performs intra-partition cross-iteration propagation.

6.2 Performance

Table 5 shows the execution times for GridGraph, LUMOS-BASE and LUMOS on h1.2xlarge. We use Highest Out-Degree

Version	TT	FT	YH
LUMOS-BASE	75.6%	67.5%	88.3%
LUMOS	81.1%	72.6%	93.8%

Table 6: Percentage of cross-iteration propagations.

to In-Degree Ratio First strategy which enables high cross-iteration propagations as shown in Table 6 (we will evaluate different partitioning strategies in Section 6.4). As we can see, LUMOS-BASE and LUMOS accelerate GridGraph in all cases. LUMOS is 1.65-1.74× faster than GridGraph while LUMOS-BASE is 1.28-1.59× faster; this is due to the reduced amount of I/O performed which gets enabled via cross-iteration propagation. Figure 6 shows the time spent in reading partitions in each case normalized w.r.t. execution time of GridGraph. As expected in out-of-core graph processing, the execution time is dominated by disk reads; for HDD we observe that GridGraph typically spends 74-83% of the time in performing disk reads, while LUMOS spends only 27-41% of the time performing disk reads compared to GridGraph. This reduction in read times results from our cross-iteration propagation technique that eliminates repetitive I/O using secondary layouts.

To study disk utilization, we vary the number of threads and measure the disk throughput and wait latencies for LUMOS. Figure 5 shows the disk throughput and wait latencies for PR on TT across 1, 2, 4, 8 and 16 threads. As we can see, the utilization is high even when using a single thread and having more threads only helps to maintain the high utilization (230-240 MB/sec) whenever the utilization drops for single thread (red trenches in Figure 5). With more threads issuing more I/O requests and utilization remaining same, the cores essentially wait more for I/O requests to complete as threads increase for HDD (shown in Figure 5b). We also observe high wait latencies as threads increase in Figure 5b due to high number of I/O requests.

It is interesting to observe that a single thread is easily able to keep the disk busy (wait times 150-200 ms) as its measured sequential read bandwidth is 278 MB/sec. Furthermore, we observe a significant dip in waiting times between ∼60-90 seconds (shown in Figure 5b) which appear while processing secondary layout; these secondary layouts are smaller, and hence the I/O requests get served quickly.

6.3 I/O Scalability

We study the impact of scaling I/O on LUMOS by setting up a RAID-0 array of 2 to 4 HDDs on d2.4xlarge, and 2 to 4 SSDs on i3.8xlarge instance. The resulting read bandwidths are shown in Table 7.

	Single Drive	RAID-0 with k drives		
		$k = 2$	$k = 3$	$k = 4$
d2.4xlarge (HDD)	195MB/s	368MB/s	590MB/s	768MB/s
i3.8xlarge (SSD)	1.2GB/s	3.8GB/s	4.1GB/s	3.9GB/s

Table 7: Sequential read bandwidth.

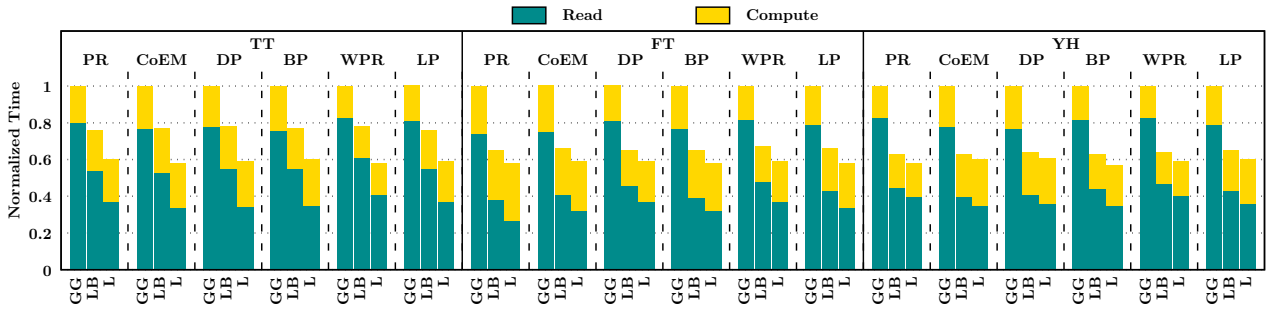


Figure 6: Read times and computation times of LUMOS and LUMOS-BASE normalized w.r.t. GridGraph's execution time.

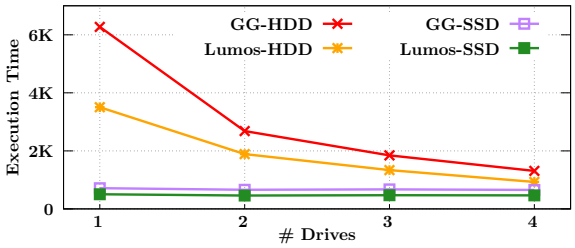


Figure 7: Execution times (in seconds) for LUMOS and GridGraph (GG) with varying number of drives. GG-HDD and LUMOS-HDD use HDDs on d2.4xlarge, while GG-SSD and LUMOS-SSD use SSDs on i3.8xlarge (see Table 7).

We ran PR on RAMT29 graph (8.6B edges) on the above setup with 64GB main memory. Figure 7 compares the execution times of LUMOS and GridGraph as number of drives increase from 1 to 4. LUMOS on d2.4xlarge scales gradually as I/O scales; it performs 1.8×, 2.6× and 3.8× faster with 2, 3 and 4 drives. It is interesting to observe that benefits of LUMOS over GridGraph diminish as number of drives increase. LUMOS performs 1.8× faster than GridGraph on a single drive and 1.3-1.4× faster on 2-4 drives. This is because GridGraph benefits from increased I/O bandwidth, which in turn leaves lesser room for effects of cross-iteration value propagation to become visible.

Contrary to HDDs, performance of LUMOS and GridGraph doesn't vary much as SSDs increase. Going from a single SSD to 2 SSDs reduces LUMOS's execution time from 505 sec to 460 sec, and the benefits of LUMOS over GridGraph also remain low with more SSDs (1.4×). This is again due to the high bandwidth provided by SSDs on i3.8xlarge (see Table 7) that alleviate I/O bottlenecks.

6.4 Partitioning Strategies

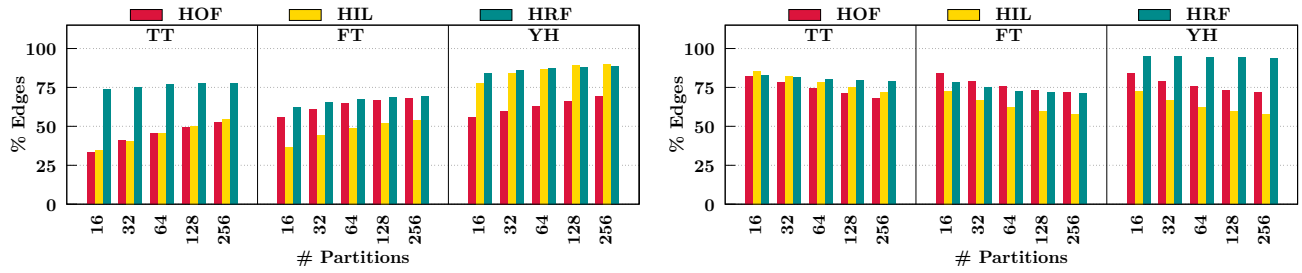
We evaluate our three light-weight partitioning strategies proposed in Section 5.3: Highest Out-Degree First (HOF), Highest In-Degree Last (HIL) and Highest Out-Degree to In-Degree Ration First (HRF). In Figure 8, we measure the amount of cross-iteration propagation for each of these strategies with and without intra-partition propagation and study sensitivity of these strategies to partition sizes.

While cross-iteration propagation depends on the structure of graph, HOF and HIL achieve 13-89% propagations across partitions whereas HRF captures the best of both and achieves significantly higher propagations (51-88%). With intra-partition propagations, all three strategies achieve significantly higher cross-iteration propagation (up to 92% for TT, 96% for FT, and 97% for YH). It is interesting to note that HOF slightly outperforms HRF and HIL for FT while HIL slightly outperforms HRF and HOF for TT; nevertheless, HRF remains useful since it achieves the middle ground between out-degree and in-degree metrics.

Finally, we observe that cross-iteration propagations across partitions increase as number of partitions increase and partitions become smaller; this is expected since edges within the same partition become potential candidates for propagations as they move to other partitions when partitions become smaller. Furthermore, when intra-partition propagation is enabled, there are fewer candidates within the same partition as number of partitions increase and hence, we observe a decreasing trend of cross-propagations. However, the scale of cross-iteration propagations remains high for HRF (70-97%) across different partition sizes, making it effective in all cases.

6.5 Preprocessing

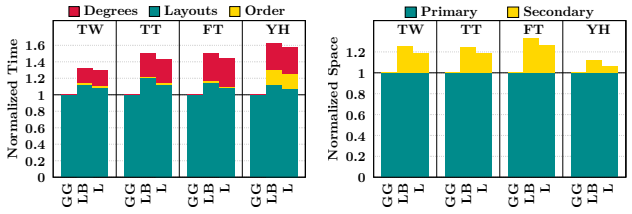
Figure 9a shows the preprocessing times normalized w.r.t. GridGraph. While our light-weight partitioning strategy requires an additional pass over edges to compute vertex degrees, the pass is lightweight since it doesn't incur simultaneous writing of edges. Furthermore, edges don't need to be sorted and vertices are ordered across buckets that determine partitions. Finally, since majority of edges enable cross-iteration propagation, secondary layouts are smaller and hence, writing them out on disk is less time consuming than that for the original graph. Figure 9b shows the increase in disk space normalized w.r.t. GridGraph; as expected, the increase is only 12-33% for LUMOS-BASE because 67-88% of edges participate in cross-iteration propagation and hence, only remainder edges are present in the secondary layouts. With intra-partition propagation, the disk space requirement increases only by 7-26%.



(a) Without locality-aware intra-partition propagation.

(b) With locality-aware intra-partition propagation.

Figure 8: Cross-iteration propagation enabled by three partitioning strategies: Highest Out-Degree First (HOF), Highest In-Degree Last (HIL), and Highest Out-Degree to In-Degree Ratio First (HRF).



(a) Preprocessing times.

(b) Disk Space.

Figure 9: Preprocessing time and disk space for LUMOS and LUMOS-BASE normalized w.r.t. GridGraph whose absolute time/space numbers are: TW = 217s/11GB; TT = 248s/15GB; FT = 334s/20GB; YH = 815s/50GB.

7 Related Work

We classify out-of-core graph processing systems into two categories based on the guarantees they provide.

(A) Synchronous Out-of-Core Graph Processing.
We discussed GridGraph [38] in Section 2; here, we briefly discuss the remaining works. GraphChi [13] pioneered single machine based out-of-core graph processing by designing partitions called *shards*, and developing a parallel sliding window model to process shards such that random disk I/O gets minimized. X-Stream [24] performs edge-centric processing using scatter-gather model. To reduce random vertex accesses, X-Stream partitions vertices and accesses edge list and update list based on partitioned vertex sets. Chaos [23] scales out X-Stream on multiple machines. FlashGraph [35] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. TurboGraph [8] is an out-of-core computation engine for graph database based on sparse matrix-vector multiplication model. Mosaic [17], GraFBoost [9] and Garaph [16] perform out-of-core processing on heterogeneous architecture containing high-bandwidth NVMe SSDs, massively parallel Xeon Phi processors, FPGAs and GPUs. Graphene [15] uses an I/O request centric graph processing model to simplify IO management by translating high-level data accesses to fine-grained IO requests. DynamicShards [31] develops dynamic partitions that eliminate unnecessary edges from partitions to reduce disk I/O.

Limitation: Since all of these works focus on computations within a single iteration, none of them leverage cross-iteration value propagation as LUMOS does. Furthermore, since these systems effectively process in partition-by-partition fashion, they can be further improved using LUMOS.

(B) Asynchronous Out-of-Core Graph Processing.
CLIP [1] exploits algorithmic asynchrony by making multiple passes over the partitions in memory. Wonderland [34] extracts effective graph abstractions to capture certain graph properties, and then performs abstraction-guided processing to infer better priority processing order and faster information propagation across graph. While the abstraction-based technique is powerful, its scope of applications is limited to path-based monotonic graph algorithms beyond which its applicability remains undefined (as mentioned in [34]). AsyncStripe [4] uses asymmetric partitioning & adaptive stripe-based access strategy to process asynchronous algorithms.
Limitation: Since synchronous guarantees are not provided by these works, their applicability is limited to asynchronous path-based algorithms. LUMOS with asynchronous processing semantics (Section 4) leverages relaxed dependencies for asynchronous algorithms as well.

Beyond Out-of-Core Graph Processing.
Google's Pregel [18], PowerGraph [6], GraphX [7], GPS [25] and Gemini [36] provide a synchronous processing model in a distributed environment, while Galois [20] and Ligra [26] offer similar guarantees in a shared memory setting. GraphBolt [19] provides synchronous processing semantics while processing streaming graphs.

8 Conclusion

We developed LUMOS, a dependency-driven out-of-core graph processing technique that performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees. Our evaluation showed that LUMOS computes future values across 71-97% of edges, hence reducing disk I/O and accelerating out-of-core graph processing by up to 1.8×.

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC*, pages 125–137, 2017.

[2] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW*, pages 595–602. ACM, 2004.

[3] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, pages 10–17, 2010.

[4] S. Cheng, G. Zhang, J. Shu, and W. Zheng. AsyncStripe: I/O Efficient Asynchronous Graph Computing on a Single Server. In *IEEE/ACM/IFIP CODES+ISSS*, page 32. ACM, 2016.

[5] Friendster network dataset. http://konect.uni-koblenz.de/networks/friendster. KONECT, 2015.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*, pages 17–30, Hollywood, CA, 2012. USENIX.

[7] 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 *USENIX OSDI*, pages 599–613, 2014.

[8] 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 *KDD*, pages 77–85, 2013.

[9] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *ISCA*. IEEE, 2018.

[10] U. Kang, D. Horng, and C. Faloutsos. Inference of Beliefs on Billion-scale Graphs. In *LDMTA*, 2010.

[11] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys*, pages 169–182. ACM, 2013.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *WWW*, pages 591–600. ACM, 2010.

[13] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX OSDI*, pages 31–46, 2012.

[14] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. MMap: Fast Billion-Scale Graph Computation on a PC via Memory Mapping. In *BigData*, pages 159–164, 2014.

[15] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *USENIX FAST*, pages 285–300, 2017.

[16] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. *USENIX ATC*, pages 195–207, 2017.

[17] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*, pages 527–543. ACM, 2017.

[18] 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 *ACM SIGMOD*, pages 135–146. ACM, 2010.

[19] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, pages 25:1–25:16. ACM, 2019.

[20] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *USENIX SOSP*, pages 456–471, 2013.

[21] K. Nigam and R. Ghani. Analyzing the Effectiveness and Applicability of Co-training. In *ACM CIKM*, pages 86–93. ACM, 2000.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[23] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *USENIX SOSP*, pages 410–424, 2015.

[24] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *USENIX SOSP*, pages 472–488, 2013.

[25] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, page 22. ACM, 2013.

[26] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN PPoPP*, pages 135–146, 2013.

[27] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] K. Vora. *Exploiting Asynchrony for Performance and Fault Tolerance in Distributed Graph Processing*. PhD thesis, University of California, Riverside, 2017.

[29] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*, pages 237–251, 2017.

[30] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *OOPSLA*, pages 861–878, 2014.

[31] K. Vora, G. H. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, pages 507–522, 2016.

[32] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Programmable and Budget-Aware Analytical Queries over Very Large Graphs on a Single PC. In *USENIX ATC*, pages 387–401, 2015.

[33] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[34] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*, pages 608–621. ACM, 2018.

[35] 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 *USENIX FAST*, pages 45–58, 2015.

[36] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX OSDI*, pages 301–316, 2016.

[37] X. Zhu and Z. Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. 2002.

[38] X. Zhu, W. Han, and W. Chen. GridGraph: Large Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*, pages 375–386, 2015.