Distributed Graph Processing System and Processing-in-memory Architecture with Precise Loop-carried Dependency Guarantee

YOUWEI ZHUO, JINGJI CHEN, GENGYU RAO, and QINYI LUO, University of Southern California, USA
YANZHI WANG, Northeastern University, USA
HAILONG YANG and DEPEI QIAN, Beihang University, China
XUEHAI QIAN, University of Southern California, USA

To hide the complexity of the underlying system, graph processing frameworks ask programmers to specify graph computations in user-defined functions (UDFs) of graph-oriented programming model. Due to the nature of distributed execution, current frameworks cannot precisely enforce the semantics of UDFs, leading to unnecessary computation and communication. It exemplifies a gap between programming model and runtime execution. This article proposes novel graph processing frameworks for distributed system and Processing-in-memory (PIM) architecture that precisely enforces loop-carried dependency; i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. Our approach instruments the UDFs to express the loop-carried dependency, then the distributed execution framework enforces the precise semantics by performing dependency propagation dynamically. Enforcing loop-carried dependency requires the sequential processing of the neighbors of each vertex distributed in different nodes. We propose to circulant scheduling in the framework to allow different nodes to process disjoint sets of edges/vertices in parallel while satisfying the sequential requirement. The technique achieves an excellent trade-off between precise semantics and parallelism—the benefits of eliminating unnecessary computation and communication offset the reduced parallelism. We implement a new distributed graph processing framework SympleGraph, and two variants of runtime systems—GraphS and GraphSR—for PIM-based graph processing architecture, which significantly outperform the state-of-the-art.

CCS Concepts: • **Computing methodologies → Distributed programming languages;**

Additional Key Words and Phrases: Graph analytics, graph algorithms, compilers, big data

Youwei Zhuo and Jingji Chen contributed equally to this work.

A preliminary version of this article (SympleGraph: Distributed Graph Processing with Precise Loop-carried Dependency Guarantee) is published in the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Compared to the published version, the new contribution is the two new PIM-based runtime and architecture GraphS and GraphSR designed based on the similar ideas that significantly outperform the state-of-the-art.

This work is supported by National Science Foundation (Grants No. CCF-1717754, No. CNS-1919289, and No. CCF-1750656).

Authors' addresses: Y. Zhuo, J. Chen, G. Rao, Q. Luo, and X. Qian (corresponding author), University of Southern California, Room 204, Ming Hsieh Department of Electrical and Computer Engineering (EEB), University Park, Los Angeles, California, 90007, USA; emails: {youweizh, jingjich, gengyura, qinyiluo, xuehai.qian}@usc.edu; Y. Wang, Northeastern University, 360 Huntington Avenue, Boston, Massachusetts, 02115, USA; email: yanz.wang@northeastern.edu; H. Yang (corresponding author) and D. Qian, Beihang University, 37 Xueyuan Rd, Haidian Qu, Beijing Shi, China; emails: {hailong.yang, depeiq}@buaa.edu.cn.

0734-2071/2021/06-ART5 $15.00
https://doi.org/10.1145/3453681

ACM Reference format:
Youwei Zhuo, Jingji Chen, Gengyu Rao, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. 2021. Distributed Graph Processing System and Processing-in-memory Architecture with Precise Loop-carried Dependency Guarantee. *ACM Trans. Comput. Syst.* 37, 1-4, Article 5 (June 2021), 37 pages.
https://doi.org/10.1145/3453681

1 INTRODUCTION

Graphs capture relationships between entities. Graph analytics has emerged as an important way to understand the relationships between heterogeneous types of data, allowing data analysts to draw valuable insights from the patterns for a wide range of real-world applications, including machine learning tasks [79], natural language processing [3, 26, 81], anomaly detection [60, 70], clustering [64, 69], recommendation [21, 29, 51], social influence analysis [15, 71, 76], and bioinformatics [2, 19, 39].

At the age of big data, the huge graph size, e.g., billions of edges, and the nature of graph computation pose significant challenges to computer system and architecture. First, the large graphs may not fit into the memory of a single machine; even if they can, the performance will be limited by the number of cores. Second, real-world graphs are typically sparse and stored in compressed representation, posing challenges for conventional memory hierarchy. Specifically, graph algorithms typically exhibit poor locality due to the random accesses in updating the neighbors; and high memory bandwidth requirement due to the small amount of computation between random accesses.

The two challenges implies that: (1) the computation capability should grow proportionally as the memory size; and (2) the computation should be conducted near where the graph data are stored. At system level, it motivates the approach to process graphs with distributed machines. At architecture level, the **Processing-in-memory (PIM)** architecture can effectively reduces data movement between memory and computation by placing computing logic inside memory dies. Though once believed to be impractical, PIM recently became a practical architecture due to the emerging 3D stacked memory technology, such as **Hybrid Memory Cube (HMC)** [16] and **High Bandwidth Memory (HBM)** [40]. The architecture is composed of multiple memory cubes. Within each memory cube, multiple DRAM dies are stacked with **Through Silicon Via (TSV)** and provide higher internal memory bandwidth up to 360 GB/s. At the bottom of the dies, computation logic (e.g., simple cores) can be embedded. Performing computation at in-memory compute logic can reduce data movements in memory hierarchy. Essentially, PIM provides "memory-capacity-proportional" bandwidth and scalability.

PIM architecture shares a *common abstraction* with distributed graph processing: graph processing can be performed with multiple *nodes* connected by certain communication links, while each node has its own local memory and computation capability. For distributed graph processing, each node corresponds to a machine with multi-cores and the memory hierarchy. In high performance clusters, the machines are connected by **Remote Direct Memory Access (RDMA)** network. The latency and bandwidth of accessing local memory are far cheaper than accessing remote memory in another machine. For PIM architectures such as HMC, each node is a memory cube, which contains stacks of high-bandwidth memory and computation logic. The memory cubes are connected by SerDes links, with 120 GB/s per link, and each cube can support up to four links. Although the total external bandwidth between memory cubes is higher than internal bandwidth, recent studies [1, 83, 86] have shown that the remote communication is still the bottleneck. In this article, we use node when describing the idea, which can be applied to both distributed system and architecture.

```
def bfs(Array[Vertex] nbr) {
  for v in V {
    for u in nbr {
      if (not visited[v] &&
       frontier[u]) {
        parent[v] = u;
        visited[v] = true;
        frontier[v] = true;
        break;
      }
    } // end for u
  } // end for v
} // end bottom_up_bfs
```

(a) Bottom-up BFS

```
def signal(Vertex v, Array[Vertex] nbr) {
  for u in nbrs {
    if (frontier[u]) {
      emit(v, u);
      break;
    }
  } // end for u
} // end signal
def slot(Vertex v, Vertex upt) {
  if (not visited[v]) {
    parent[v] = upt;
    visited[v] = true;
    frontier[v] = true;
  }
} // end slot
```

(b) Bottom-up BFS in Gemini

Fig. 1. Bottom-up BFS algorithm.

To hide the detail and complexity of distributed system and architecture, the common approach is to build a graph processing framework that executes programs written in the graph oriented programming models (APIs). A number of distributed graph processing frameworks have been proposed, e.g., Pregel [45], GraphLab [44], PowerGraph [24], D-Galois [18], and Gemini [85]. These frameworks partition the graph to distributed memory, so the neighbors of a vertex are assigned to different machines. Similarly, the distributed PIM-based architectures, e.g., Tesseract [1], GraphP [83], GraphQ [86], and GraphH [17], also support graph algorithms with a runtime framework that handles data in different memory cubes. To achieve the efficient implementations, the APIs of frameworks for distributed machines and memory cubes are slightly different. The details will be discussed in Sections 2.2 and 2.3.

In essence, the frameworks can be considered as the *interface* between algorithms and system/architecture. The graph computation is abstracted as vertex-centric ***User-Defined Functions (UDFs)*** $P(v)$, which is executed for each vertex v. In each $P(v)$, programmers can access the neighbors of v as if they are local, this is why the domain-specific APIs can simplify programming. The framework is responsible for *transparently* distributing the function to different nodes, scheduling the computations and communication, performing synchronization, and ensure that the distribute execution output correct results. To achieve good performance, both communication and computation need to be efficient. The communication problem, which is closely related to graph partition and replication, has been traditionally a key consideration. Prior works have proposed 1D [44, 45], 2D [24, 85], 3D [82] partition, and investigate the design space extensively [18]. This article makes the first attempt to improve the efficiency of the two factors at the same time by reducing redundant computation and communication, by enforcing the precise dependency among UDFs.

Loop-carried dependency is a common code pattern used in UDFs: when traversing the neighbors of a vertex in a loop, a UDF decides whether to break or continue, based on the state of processing previous neighbors. Specifically, consider two neighbors u_1 and u_2 of vertex v. If u_1 satisfies an algorithm-specific condition, then u_2 will not be processed due to the dependency. The pattern appears in several important algorithms. Consider the bottom-up **breadth-first search (BFS)** [7] with pseudocode in Figure 1(a). In each iteration, the algorithm visits the neighbors of "unvisited" vertices. If *any* of the neighbors of the current unvisited vertex is in the "frontier," then it will no longer traverse other neighbors and mark the vertex as "visited."

In distributed frameworks [14, 18, 23, 24, 32, 34, 45, 62, 78, 85], programmers can write a control flow with the break statement in UDF to indicate the control dependency. Figure 1(b) shows signal-slot implementation of bottom-up BFS in Gemini [85]. The `signal` and `slot` UDF specify the computation to process each neighbor of a vertex and vertex update, respectively. We see

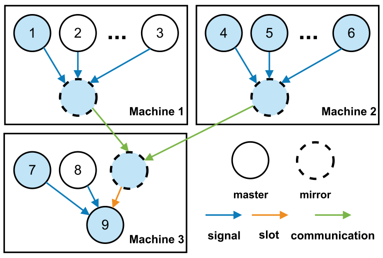


Fig. 2. Bottom-up BFS execution.

that the bottom-up BFS UDF has *control dependency*. The signal function iterates the neighbors of vertex v, and breaks out of the loop when it finds the neighbor in the frontier (Line 5). This control dependency expresses the semantics of skipping the following edges and avoids unnecessary edge traversals. However, if u_1 and u_2 are distributed in different machines, then u_1 and u_2 can be processed in parallel and u_2 does not know the state after processing u_1. Therefore, the loop-carried dependency specified in UDF is not precisely enforced in the execution, thereby only an "illusion." We will show in Section 2.3 that the frameworks for PIM-based architecture incur the similar problem.

The consequence of such imprecise execution behavior is *unnecessary computation and communication*. As shown in Figure 2, vertex 9 has eight neighbors, two of them (vertex 7 and 8) are allocated in machine 3, the same as the *master* copy of vertex 9. The others are allocated in machine 1 and 2. More background details on graph partition will be discussed in Section 2.2. To perform the `signal` UDF in remote machines, *mirrors* of vertex 9 are created. *Update* communication is incurred when mirrors (machine 1 and 2) transfer partial results of `signal` to the master of vertex 9 (machine 3). Unnecessary computation is incurred when a mirror performs computations on vertex 9's neighbors while the condition has already been satisfied. Unnecessary update communication is incurred when the mirror sends partial results to the master.

To address this problem, we propose the novel graph processing frameworks for distributed system and PIM architecture that precisely enforces loop-carried dependency, i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. Our approach analyzes the UDFs of unmodified codes, identifies, and instruments UDF to express the loop-carried dependency. The distributed framework enforces the dependency semantics by performing dynamic dependency propagation. Specifically, a new type of *dependency communication* propagates dependency among mirrors and back to master. Existing frameworks only support *update communication*, which aggregates updates from mirrors to master.

Enforcing loop-carried dependency requires that all neighbors of a vertex are processed sequentially. To enable sufficient parallelism while satisfying the sequential requirement, we propose *circulant scheduling* and divide the execution of each iteration into *steps*, during which different nodes—either machines or memory cubes—process disjoint sets of edges and vertices. If one node determines that the execution should break in a step, then such break information is passed to the following nodes so that the remaining neighbors are not processed. In practice, the computation and update communication of each step can be overlapped, thus the fine-grained steps do not introduce much extra overhead.

An important benefit of the idea is that it not only eliminates unnecessary computation but potentially reduces the total amount of communication. On the one side, small dependency messages that are organized as a bit map (one bit per vertex) circulating around all mirrors and master, do not exist in current frameworks, and thus incur extra communication. On the other side, precisely

```
def mis(Array[Vertex] nbr) {
  for v in V {
    flag = true;
    for u in nbr {
      if (active[u] &&
      color[u] < color[v]) {
        flag = false;
        break;
      }
      if (flag)
        is_mis[v] = true;
    } // end for u
  } // end for v
} // end mis
```

(a) MIS

```
def kcore(Array[Vertex] nbr) {
  for v in V {
    cnt = 0;
    for u in nbr {
      if (active[u]) {
        cnt += 1;
        if (cnt >= k) {
            active[v] = true;
            break;
        }
      }
    } // end for u
    active[v] = false;
  } // end for v
} // end kcore
```

(b) K-core

```
def kmeans(Array[Vertex] nbr) {
  // generate C random centers
  for v in V {
    for u in nbr {
        if (assigned_to_cluster[u])
              {
            cluster[v] = cluster[u
                  ];
            break;
        }
    } // end for u
  } // end for v
} // end kmeans
```

(c) K-means

```
def sample(Vertex v, Array[
        Vertex] nbr) {
  // generate C random number
  r = rand()
  // set prefix-sum
  weight = 0
  for u in nbr {
    weight += weight[u]
    if (weight >= r) {
        select[u] = true;
        break;
    }
  } // end for u
} // end kmeans
```

(d) Graph Sampling

Fig. 3. Examples of algorithms with loop-carried dependency.

enforcing loop-carried dependency can eliminate unnecessary computation and update communication. Our results show that the total amount of communication is indeed reduced in most cases. To further reduce dependency communication, we can apply *differentiates* dependency communication for high-degree and low-degree vertices, and only perform dependency propagation for high-degree vertices.

Based on the ideas, we implemented a new distributed graph processing framework SympleGraph, and two variants of runtime systems—GraphS and GraphSR—for PIM-based graph processing architecture. SympleGraph is based on the `signal` and `slot` APIs. We apply double buffering in SympleGraph to enable computation and dependency communication overlapping and alleviate load imbalance. GraphS and GraphSR provide the `GenUpdate` (similar to `signal`) and `ApplyUpdate` (similar to `slot`) APIs but they are executing on a PIM-based architecture with very different computation and communication capacity and trade-off. We present the runtime implementation that carefully handles the synchronization between memory cubes and batched communication. Moreover, GraphSR further improves the performance for certain algorithms by accumulating partial results.

To evaluate SympleGraph, we conduct the experiments on three clusters using five algorithms and four real-world datasets and three synthesized scale-free graphs with R-MAT generator [13]. We compare SympleGraph with two state-of-the-art distributed graph processing systems, Gemini [85] and D-Galois [18]. The results show that SympleGraph significantly advances the state-of-the-art, outperforming Gemini and D-Galois on average by 1.42× and 3.30×, and up to 2.30× and 7.76×, respectively. The communication reduction compared to Gemini is 40.95% on average, and up to 67.48%.

We evaluate GraphS and GraphSR with a zSim [63]-based simulator using real-world graphs as large as Friendster [41] the same algorithms. They are *not* commonly used in recent works on graph processing architecture. We compare GraphS and GraphSR to GraphQ, the state-of-the-art PIM-based graph processing architecture. The results show that GraphS achieves on average 2.2× (maximum 4.37×) speedup, on average 32.8% (maximum 50.5%) inter-cube communication reduction. They lead to 51.6% energy saving on average. With partial results propagation, GraphSR achieves on average 12.5× (maximum 19.91×) speedup, on average 90.23% (maximum 97.79%) inter-cube communication reduction. They lead to 91.4% energy saving on average.

2 BACKGROUND AND PROBLEM FORMALIZATION

2.1 Graph and Graph Algorithm

A graph G is defined as (V, E) where V is the set of vertices, and E is the set of edges (u, v) (u and v belong to V). The neighbors of a vertex v are vertices that each has an edge connected to v. The degree of a vertex is the number of neighbors. In the following, we explain five important iterative graph algorithms whose implementations based on vertex functions will incur loop-carried dependency in UDF. Figure 3 shows the pseudocode of one iteration of each algorithm in sequential implementation.

Breadth-First Search. It is an iterative graph traversal algorithm that finds the shortest path in an unweighted graph. The conventional BFS algorithm follows the top-down approach: BFS first visits a root vertex, then in each iteration, the newly "visited" vertices become the "frontier" and BFS visits all the neighbors of the "frontier".

Bottom-up BFS [7] changes the direction of traversal. In each iteration, it visits the neighbors of "unvisited" vertices, if one of them is in the "frontier," the traversal of other neighbors will be skipped, and the current vertex is added to the frontier and marked as "visited." Compared to the top-down approach, bottom-up BFS avoids the inefficiency due to multiple visits of one new vertex in the frontier and significantly reduces the number of edges traversed.

Maximal Independent Set. An independent set is a set of vertices in a graph, in which any two vertices are non-adjacent. A **Maximal Independent Set (MIS)** is an independent set that is not a subset of any other independent set. A heuristic MIS algorithm (Figure 3(a)) is based on graph coloring. First, each vertex is assigned distinct values (colors) and marked as active. In each iteration, we find a new MIS composed of active vertices with the smallest color value among their active neighbors' colors. The new MIS vertices will be removed from further execution (marked as inactive).

K-core. A K-core of a graph G is a maximal subgraph of G in which all vertices have a degree at least k. The standard K-core algorithm [67] (Figure 3(b))[1] removes the vertices that have a degree less than K. Since removing vertices will decrease the degree of its neighbors, the operation is performed iteratively until no more removal is needed. When counting the number of neighbors for each vertex, if the count reaches K, we can exit the loop and mark this vertex as "no remove."

K-means. It is a popular clustering algorithm in data mining. Graph-based K-means [62] is one of its variants where the distance between two vertices is defined as the length of the shortest path between them (assuming that the length of every edge is one). The algorithm shown in Figure 3(c) consists of four steps: (1) Randomly generate a set of cluster centers. (2) Assign every vertex to the nearest cluster center. (3) Calculate the sum of distance from every vertex to its belonging cluster center. (4) If the clustering is good enough or the number of iterations exceed some pre-specified threshold, then terminate the algorithm; else, go to step (1) and repeat the algorithm.

Graph Sampling. It is an algorithm that picks a subset of vertices or edges of the original graph. We show an example of neighbor vertex sampling in Figure 3(d), which is the core component of graph machine learning algorithms, such as DeepWalk [58], node2vec [28], and Graph Convolutional Networks [6]. To sample from the neighbor of the vertex based on weights, we need to generate a uniform random number and find its position in the prefix-sum array of the weights, i.e., the index in the array that the first prefix-sum element is larger than or equal to our random number.[2]

[1]There are other K-core algorithms with linear time complexity [47]. We choose this algorithm to demonstrate the basic code pattern. We also compare with the algorithm in evaluation.
[2]There are other sampling algorithms, such as the alias method. It builds alias table step to exhibit a similar pattern that searches prefix-sum array. We choose this algorithm, since it reflects our basic code pattern.

```
// mirror signal
for m in machines {
    for mirror in m.mirrors(v) {
        signal(v, nbrs(v));
    }
}
// master slot
for (v, update) in signals {
    slot(v, update);
}
```

Fig. 4. Signal-slot in pull mode.

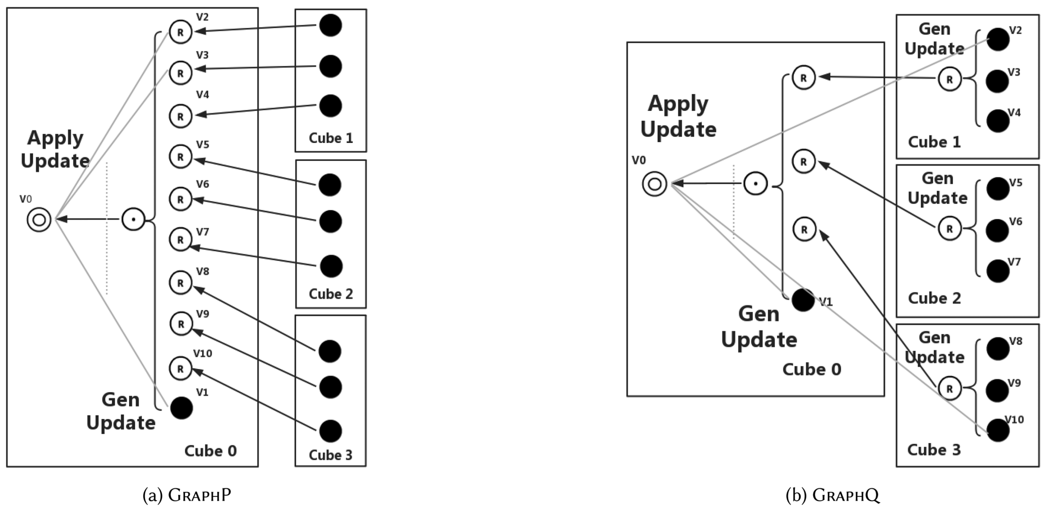
2.2 Distributed Graph Processing Frameworks

There are two design aspects of distributed graph framework: programming abstraction, and graph partition/replication. Programming abstraction deals with how to express algorithms with a vertex function. Graph partition determines how vertices and edges are distributed, replicated, and synchronized in different machines.

Master-mirror. To describe vertex replications, current frameworks [14, 18, 24, 85] adopt the *master-mirror* notion: each vertex is owned by one machine, which keeps the *master* copy, its replications on other machines are *mirrors*. The distribution of masters and mirrors is determined by graph partition. There are three types of graph partition techniques based on the definition in Reference [18]. *Incoming edge-cut*: Incoming edges of one vertex are assigned only to one machine, while its outgoing edges may be partitioned; *Outgoing edge-cut*: Outgoing edges of each vertex are assigned only to one machine, while its incoming edges are partitioned. It is used in several systems, including Pregel [45], GraphLab [44], and Gemini [85]. *Vertex-cut*: Both the outgoing and incoming edges of a vertex can be assigned to different machines. It is used in PowerGraph [24] and GraphX [25]. Recent work [82] also proposed 3D graph partition that divides the vector data of vertices into layers. This dimension is orthogonal to the edge and vertex dimensions considered in other partitioning methods.

We build SympleGraph based on Gemini, the state-of-the-art distributed graph processing framework using outgoing edge-cut partition. However, our ideas also apply to vertex-cut and other distributed frameworks. It is not applicable to incoming edge-cut, which will be discussed in Section 2.4. In outgoing edge-cut, a mirror vertex is generated if its incoming edges are partitioned among multiple machines. Figure 2 shows an example of a graph distributed in three machines. Circles with solid lines are masters, and circles with dashed lines are mirrors. Here, vertex 9 has 8 incoming edges, i.e., sources vertex 1 to 8. Machine 1 contains the master of vertex 1 to 3, and machine 2 contains the master of vertex 4 to 6. The master of vertex 9 resides on machine 3 but its incoming edges are partitioned across all three machines, so mirrors of v are created on machine 1 and 2.

Signal-slot. Ligra [68] discusses the two modes of signal-slot: push and pull. Push mode traverses and updates the outgoing neighbors of vertices, while pull mode traverses the incoming neighbors. The five graph algorithms discussed earlier are more efficient in pull mode in most iterations, and SympleGraph optimization focuses on pull mode. Figure 4 shows the pseudocode of pull mode. The signal function is first executed on mirrors *in parallel.* The mirrors then send update messages to the master machine. On receiving an update message, the master machine applies the slot function to aggregate the update, and then eventually updates master vertex after receiving all updates. Figure 2 also illustrated how the signal-slot function is applied for vertex 9. The blue edges



(a) GraphP (b) GraphQ

Fig. 5. GraphP and GraphQ.

(in machine 1 and 2) refer to signals, and the yellow edges (in machine 3) refer to slots. Green edges across machines indicate communication.

2.3 PIM-based Graph Processing

PIM architecture reduces data movements by performing computations close to where the data are stored. 3D memory technologies (e.g., HMC [16] and HBM [40]) make PIM feasible by integrating memory dies and compute logic in the same package, achieving high memory bandwidth and low latency. Similar to recent works [1, 83, 86], this article considers a general PIM architecture that captures key features of specific PIM implementations. The architecture is composed of multiple *cubes* connected by external links (e.g., SerDes links in HMC with 120GB/s per link). Within each cube, multiple DRAM dies are stacked with TSV and provide higher internal memory bandwidth up to 320 GB/s. At the bottom of the dies, computational logics (e.g., simple cores) could be embedded. In Tesseract [1], a small single-issue in-order core is placed at the logic die of each vault. It is feasible, because the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm^2 for each core [4]) corresponds to only 9.6% of the area of an 8 Gb DRAM die area (e.g., 226 mm^2 [66]). GraphS assumes the same setting. With 16 cubes, the whole system delivers 5 TB/s memory bandwidth, considerably larger than conventional memory systems. Moreover, the memory bandwidth grows proportionally with capacity in a scalable manner.

All PIM-based graph processing architectures partition the graph into subgraphs that can fit into each cube. The connectivity of the subgraphs leads to inter-cube communication. GraphP [83] shows that communication is affected by the graph partition strategy and proposes a method to reduce the communication from one inter-cube message per cross-cube edge in Tesseract [1] to one inter-cube message per replica synchronization. This is enabled by assigning disjoint edge sets to different cubes and generate replicas[3] when a vertex is connected to two edges in different cubes. Evaluation results show that it can reduce inter-cube communication by at least 90% in all experiments with broadcast optimization. Figure 5(a) illustrates the insights of GraphP. ● and ◎

[3]The term replica is conceptually the same as mirror in Gemini. In the context of PIM-based graph processing, it is consistent with the existing literature.

represent source and destination master vertices, respectively. ® represents the replica of a vertex. Each vertex can have one master and multiple replicas in other cubes depending on graph partition. In the example, v_0 has 10 neighbors, all edges connected to v_0 are assigned to cube 0, only one v_1 is allocated in the same cube. Since the master of the others $v_2 \sim v_{10}$ are allocated in remote cube 1 ∼ cube 3, their replicas are created in cube 1. When the masters of $v_2 \sim v_{10}$ are updated, the runtime system will generate inter-cube messages to synchronize the replicas, indicated by the arrows between cubes. It also shows how the two APIs are executed: `GenUpdate` generates the update for a vertex based on all its neighbors; `ApplyUpdate` updates the property of a vertex, at the same time the runtime system updates all its replicas.

GraphQ [86] enables regular and batched communication by generating inter-cube messages between a specific cube pair together. Since APIs and runtime system are decoupled, GraphQ can also support `GenUpdate` and `ApplyUpdate`. Since all communication from cube i and cube j will happen together in batch, GraphQ naturally performs partial `GenUpdate` in the sender before they are transferred to the receiver cube. Thus, we have replicas of the destination (v_0) in cube 1 ∼ cube 3. As shown in Figure 5(b), when partially reduced values are received in cube 0, they can be reduced to a single value and apply to the master of v_0 by `ApplyUpdate`. Thus, `GenUpdate` function is executed in both local and remote cubes. Based on the above explanation, GraphQ supports GraphP's APIs (which can express loop-carried dependency) but enables more efficient execution than GraphP, we use GraphQ with `GenUpdate` and `ApplyUpdate` as the baseline. We can see that they are equivalent to `signal` and `slot` in Gemini. Thus, the proposed ideas can be implemented similarly. However, the runtime framework of GraphS and GraphSR have to be built based on the architecture primitives for synchronization and efficient batched communication.

2.4 Inefficiencies with Existing Frameworks

We can formalize the signal-slot abstraction by borrowing the notions of distributed functions in Reference [80].

Definition 2.1. We use u to denote a sequence of neighbors of vertex v, and use $u_1 \oplus u_2$ to denote the concatenation of u_1 and u_2. A function H is *associative-decomposable* if there exist two functions I and C satisfying the following conditions:

(1) H is the composition of I and C: $\forall u,\ H(u) = C(I(u))$;
(2) I is commutative: $\forall u_1, u_2,\ I(u_1 \oplus u_2) = I(u_2 \oplus u_1)$;
(3) C is commutative: $\forall u_1, u_2,\ C(u_1 \oplus u_2) = C(u_2 \oplus u_1)$;
(4) C is associative: $\forall u_1, u_2,\ u_3, C(C(u_1 \oplus u_1) \oplus u_3) = C(u_1 \oplus C(u_2 \oplus u_2))$.

Generally, common graph algorithms can be represented by associative-decomposable vertex functions in Definition 2.1. Intuitively, I and C correspond to signal and slot functions. Note that the abstraction specification is also a system implementation specification. If C is commutative and associative, then a system can can perform C efficiently: the execution can be out-of-order with partial aggregation.

However, this essentially means that existing distributed systems require the graph algorithms to satisfy a stronger condition.

Definition 2.2. A function H is *parallelized* associative-decomposable if there exist two functions I and C satisfying the conditions of Definition 2.1, and I preserves concatenation in H:

$$\forall u_1, u_2,\ H\left(u_1 \oplus u_2\right) = C(I(u_1 \oplus u_2)) = C\left(I(u_1) \oplus I(u_2)\right).$$

Gemini and other existing frameworks require the graph algorithms to satisfy Definition 2.2, which offers parallelism and ensures correctness. On the one hand, Gemini can distribute the execution of neighbors to different machines, and perform I independently and in parallel. On the other hand, the output of H is the same as if executing I sequentially.

Existing frameworks are designed for algorithms without loop-carried dependency. We first define loop-carried dependency and dependent execution. After that, we can rewrite Definition 2.1 as Definition 2.4.

Definition 2.3. We use $I(u_2|u_1)$ to denote $I(u_2)$ given the state that $I(u_1)$ has finished, such that $\forall u_1, u_2,\ I(u_1 \oplus u_2) = I(u_1) \oplus I(u_2|u_1)$. A function I has no loop-carried dependency if $\forall u_1, u_2,\ I(u_2|u_1) = I(u_2)$.

Definition 2.4. A function H is associative-decomposable if there exist two functions I and C satisfying the conditions of Definition 2.1. H has the property:

$$\forall u_1, u_2,\ H(u_1 \oplus u_2) = C(I(u_1 \oplus u_2)) = C(I(u_1) \oplus I(u_2|u_1)).$$

By Definition 2.3, these algorithms always satisfy both Definition 2.4 and Definition 2.2. Otherwise, if a graph algorithm only satisfies Definition 2.4, but not Definition 2.2, existing frameworks will not output the correct results. Fortunately, many graph algorithms with loop-carried dependency (including the five algorithms in this article) satisfy Definition 2.2, so correctness is not an issue for existing frameworks.

However, the intermediate output of I can be different. By Definition 2.2, we will execute $I(u_1)$ and $I(u_2)$. By Definition 2.4, if we enforce dependency, we will execute $I(u_1)$ and $I(u_2|u_1)$. The difference comes down to $I(u_2)$ and $I(u_2|u_1)$. If we use $cost(\cdot)$ to denote the computation cost of a function or the communication amount for the output of a function, then a function I has redundancy without enforcing dependency if $\forall u_1, u_2,\ cost(I(u_2)) \geq cost(I(u_2|u_1)))$ and $\exists u_1, u_2,\ cost(I(u_2)) > cost(I(u_2|u_1))$.

We can define functions with break semantics:

$$\exists u_1, u_2,\ I(u_2|u_1) = I(\varnothing) = \varnothing.$$

The computation cost for $I(\varnothing)$ is 0, and the communication cost for $\varnothing$ is 0. It is evident that these functions suffer from the redundancy problem. We can use bottom-up BFS and Figure 2 as an example to calculate the cost. The computation cost is the number of edges traversed and the communication cost is the update message to the master. For now, we ignore the overhead of enforcing dependency. The circles with colors are incoming neighbors that will trigger the break branch. On machine 1, the signal function breaks traversing after vertex 1, so vertex 2 and vertex 3 are skipped. On machine 2, it iterates all 3 vertices if machine 2 is not aware of the dependency in machine 1. The computation cost is 4 edges traversed (the sum of machine 1 and machine 2), and the communication is 2 messages (1 message from each machine). However, if we enforce the dependency, then all vertices in machine 2 should not have been processed. The computation cost is 1 edge traversed (only on machine 1) and the communication is 1 message (only from machine 1).

In summary, a graph algorithm with loop-carried dependency can be *correct* in existing frameworks, if it satisfies Definition 2.2. However, it can be *inefficient* with both redundant computation, and communication when loop-carried dependency is not faithfully enforced in a distributed environment.

Applicability. The problem exists for all graph partitions except the incoming edge-cut, i.e., all of the incoming edges of one vertex are on the same machine, and the execution of UDFs is not distributed to remote machines. To our knowledge, none of distributed systems [14, 18, 23, 24, 32, 34, 45, 62, 78, 85] precisely enforce loop-carried dependency semantics. While the incoming

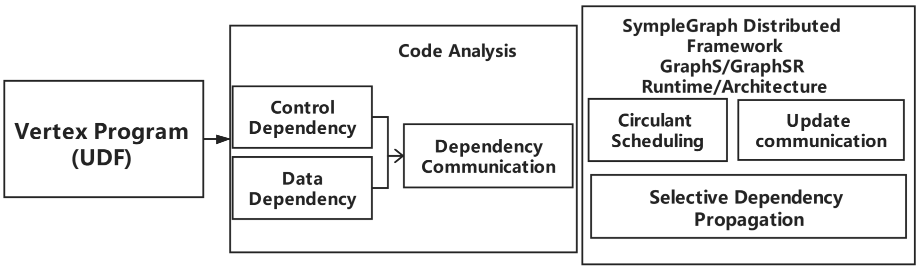


Fig. 6. Overall of system and architecture workflow.

edge-cut is an exception, the partition is inefficient and rarely used due to load imbalance issues. According to D-Galois (Gluon), they used the vertex-cut partition by default, “since it performs well at scale” [18].

The problem exists for many algorithms with loop-carried dependency. For the other four graph algorithms discussed in Section 2.1: **MIS** has *control dependency*. If one vertex already finds itself not the smallest one, then it will not be marked as a new MIS in this iteration and thus break out of the neighbor traversal. **K-core** has *data and control dependency*. If the vertex has more than K neighbors, then it will not be marked as removed in this iteration, and further computation can be skipped. **K-means** has *control dependency*: when one of the neighbors is assigned to the nearest cluster center, the vertex can be assigned with the same center. **Graph sampling** has *data and control dependency*. The sample is dependent on the random number and all the preceding neighbors’ weight sum. It exits once one neighbour is selected. Note that we use these algorithms as typical examples to demonstrate the effectiveness of our idea. They all share the basic code pattern, which can be used as the building blocks of other more complicated algorithms.

3 OVERVIEW OF SYSTEM AND ARCHITECTURE

Our goal is to build new distributed graph processing frameworks and runtime systems that precisely enforces loop-carried dependency semantics in UDFs. We propose a workflow that consists of two components. The first one is *UDF code analysis*, which (1) determines whether the UDF contains loop-carried dependency; (2) if so, identifies the dependency state that need to be propagated during the execution; and (3) instruments codes of UDF to insert dependency communication codes executed by the framework to enforce the dependency across distributed nodes. The second component is system or architecture/runtime supports for loop-carried dependency on the analyzed UDF codes and communication optimization. The key technique is *dependency communication*, which *propagates dependency among mirrors* and back to master. To enforce dependency correctly, for a given vertex, execution of UDF related to its neighbors assigned to different nodes must be performed *sequentially*. The key challenge is how to *enforce the sequential semantics while still enabling enough parallelism*? We solve this problem by circulant scheduling and other communication optimizations. The overall workflow of the proposed system and architecture is shown in Figure 6.

4 USER-DEFINED FUNCTION ANALYSIS

4.1 Instrumentation Primitives

We propose dependency communication primitives, which are used internally inside the framework and transparent to programmers. In the following discussion, we assume that the

```
1  struct DepBFS : DepMessage { // datatype
2    bit skip?;
3  };
4  def signal(Vertex v, Array[Vertex] nbrs) {
5    DepBFS d = receive_dep(v); // new code
6    if (d.skip?) {
7      return;
8    }
9    for u in nbrs(v) {
10     if (frontier[u]) {
11       emit(v, u);
12       emit_dep(v, d); // new code
13       break;
14     }
15   }
16 }
17 def slot(Vertex v, Vertex upt) {
18     ... // no changes
19 }
```

Fig. 7. Instrumented bottom-up BFS UDFs.

instrumentation is performed to the programs on Gemini described `signal` and `slot`. In Section 6, we describe the similar procedure for GraphS and GraphSR runtimes. Dependent message has a data type `DepMessage` with two types of data members: a bit for control dependency, and data values for data dependency. To enforce loop-carried dependency, the relevant UDFs need to be executed sequentially. Two functions `emit_dep<T>` and `receive_dep<T>` send and receive the dependency state of a vertex, where the type of `T` is `DepMessage`. We first describe how SympleGraph uses these primitives in the instrumented codes. Shortly, we will describe the details of SympleGraph analyzer to generate the instrumented codes.

Figure 7 shows the analyzed UDFs of bottom-up BFS with dependency information and primitive. When processing a vertex u, the framework first executes `emit_dep` to get whether the following computation related to this vertex should be skipped (Lines 5 ∼ 7). After the vertex u is added to the current frontier, `emit_dep` is inserted to notify the next machine, which executes the function. Note that `emit_dep` and `emit_dep` do not specify the sender and receiver of the dependency message, it is intentional as such information is pre-determined by the framework to support circulant scheduling.

4.2 Code Analysis

To implement the dependent computation of function I in Definition 2.4, we instrument I to include dependency communication and leave C unchanged. We develop SympleGraph analyzer , a prototype tool based on Gemini's signal-slot programming abstraction. To simplify the analyzer design, we make the following assumptions on the UDFs.

- The UDFs store dependency data in capture variables of lambda expressions. Copy statements of these variables are not allowed so that we can locate the UDFs and variables.
- The UDFs traverse neighbor vertices in a loop.

Based on the assumptions, we design SympleGraph analyzer as two passes in clang LibTooling at clang-AST level.

(1) In the first pass, our analyzer locates the UDFs and analyzes the function body to determine whether loop-carried dependency exists.

(a) Use clang-lib to compile the source code and obtain the corresponding Clang-AST.
(b) Traverse the AST to: (1) locate the UDF; (2) locate all process-edges (sparse-signal, sparse-slot, dense-signal, dense-slot) calls and look for the definitions of all dense-signal functions; (3) search for all for-loops that traverse neighbors in dense-signal functions and check whether loop-dependency patterns exist (there is at least one break statement related to the for-loop); (4) store all AST nodes of interests;

(2) In the second pass, if the dependency exits, it identifies the dependency state for communication and performs a source-to-source transformation.
(a) Insert dependency communication initialization code.
(b) Before the loop in UDF, insert a new control flow that checks dependency in preceding loops with `receive_dep`.
(c) Inside the loop in UDF, insert `emit_dep` before the corresponding break statement to propagate the dependency message.

Based on the codes in Figure 1(b), SympleGraph analyzer will generate the source codes in Figure 7.

4.3 Discussion

In this section, we discuss the alternative approaches that can be used to enforce loop-carried dependency.

New Graph Domain Specific Language. Besides the analysis, SympleGraph provides a new **Domain Specific Language (DSL)** and asks the programmer to express loop dependency and state. We support a new functional interface *fold_while* to replace the for-loop. It specifies a state machine and takes three parameters: initial dependency data, a function that composes dependency state and current neighbor, a condition that exits the loop. The compiler can easily determine the dependency state and generate the corresponding optimized code.

Manual analysis and instrumentation. Some will argue that if graph algorithms UDFs are simple enough, the programmers can manually analyze and optimize the code. SympleGraph also exposes communication primitives to the programmers so that they can still leverage the optimizations when the code is not amendable to static analysis.

Manual analysis may even provide more performance benefits, because some optimizations are difficult for static analysis to reason about. One example is the communication buffer. In bottom-up BFS, users can choose to *repurpose* "visited" array as the break dependency state. The "visited" is a bit vector and can be implemented as a bitmap. When we record the dependency for a vertex, the "visited" has already been set, so we can reduce computation by avoiding the bit set operation in the dependency bitmap. When we send the dependency, we can actually send "visited" and avoid the memory allocation for dependency communication.

However, writing such optimizations manually is not recommended for two reasons. First, the optimizations in memory footprint and computation are not the bottleneck to the overall performance. The memory reduction is one bit per vertex, while in every graph algorithm, the data field of each vertex takes at least four bytes. As for the computation reduction, setting a bit sequentially in a bitmap is also negligible compared with the random edge traversals. In our evaluation, the performance benefit is not noticeable (within 1% in execution time). Second, manual optimizations will affect the readability of the source code, and increase the burden of the user, hurting programmability. It contradicts the original purpose of domain-specific systems. The programmers need to have a solid understanding of both the algorithm and the system. In the same example, there is another bitmap "frontier" in the algorithm. However, it is incorrect to repurpose "frontier" as the dependency data.

```
for v in V {
    // mirror signal
    for m in machines {
        for mirror in m.mirrors(v) {
            ...
            emit(v, upt) // update
            emit_dep(v, dep) // dependency
        }
    }
}
```

Fig. 8. Circulant Scheduling.

5 SYMPLEGRAPH SYSTEM

In this section, we discuss how SympleGraph schedules dependency communication to enforces dependent execution and several system optimizations.

5.1 Enforcing Dependency: Circulant Scheduling

By expanding the signal expressions in Figure 4 for all vertices, we have Figure 8, a *nested loop*. Our goals are to (1) parallelize the outer loop and (2) enforce the dependency order of the inner loop. However, if each vertex starts from the same machine, then the other machines are idle and parallelism is limited. To preserve parallelism and enforce dependency simultaneously, we have to schedule each vertex to start with mirrors from different machines. We formalize the idea as *circulant scheduling*, which divides the iteration into p steps for p machines and execute I according to a circulant permutation. In fact, any cyclic permutation will work, and we choose one circulant for simplicity.

Definition 5.1 (Circulant Scheduling). A circulant permutation σ is defined as $\sigma(i) = (i+p-1)\%p$, and initially $\sigma(i) = i, i = 0, \ldots, (p-1)$. The vertices in a graph is divided into p disjoint sets according to he master vertices. Let $u^{(i)}$ denote the sequence of neighbors of master vertices on machine i. In step j ($j = 0, 1, \ldots, p-1$), circulant scheduling executes $I(u^{(i)})$ on machine $\sigma^j(i)$.

Circulant scheduling achieves the two goals and the correctness can be inferred from the properties of permutation. For any specific vertex set, its execution follows the order of $I(u_{\sigma^{j-1}} | u_{\sigma^0} \oplus u_{\sigma^1} \oplus \ldots \oplus u_{\sigma^{j-2}})$, starting from step 0. For any specific step j, the scheduling specifies different machines, because σ^j is a permutation. For example, the permutation of step 0 based on $(0, 1, 2, 3)$ is $\sigma^0 = (3, 0, 1, 2)$. In step 0 (the first step), $I(u^{(0)})$ (the sequence of neighbors of master vertices on machine 0) is processed on machine 3 ($\sigma^0(0) = 3$). In step 1 (the second step), $\sigma^1 = (2, 3, 0, 1)$, $I(u^{(0)})$ is processed on machine 2 ($\sigma^1(0) = 2$).

Figure 9 shows an example with four machines. Figure 9 (a) shows the matrix view of the graph. An element (i,j) in the matrix represents an edge ($v_i \leftarrow v_j$). Similarly, we use the notion $[i, j]$ to represent a subgraph with edges from machine i to machine j. Based on circulant scheduling, machine 0 first processes all edges in $[0, 1]$ and then $[0, 2]$, $[0, 3]$, $[0, 0]$. $[0, 1]$ contains the edges between master vertices on machine 1 and their neighbors in machine 0. The other machines are similar. In the same step, each machine i processes edges in *different* subgraph $[i, j]$ *in parallel*. For example, in step 0, the subgraphs processed by machine 0, 1, 2, 3 are $[0, 1]$,$[1, 2]$,$[2, 3]$,$[3, 0]$, respectively. After all steps, edges in $[j, i]$, $j \in \{0, 1, 2, 3\}$, are processed *sequentially*.

Figure 9(b) shows the step execution according to Figure 9(a) with dependency communication. The dependency communication pattern is the *same* for all steps: each machine only communicates with the machine on its left. Note that circulant scheduling enables more parallelism, because each machine processes disjoint sets of edges in parallel. It is still more restrictive than arbitrary

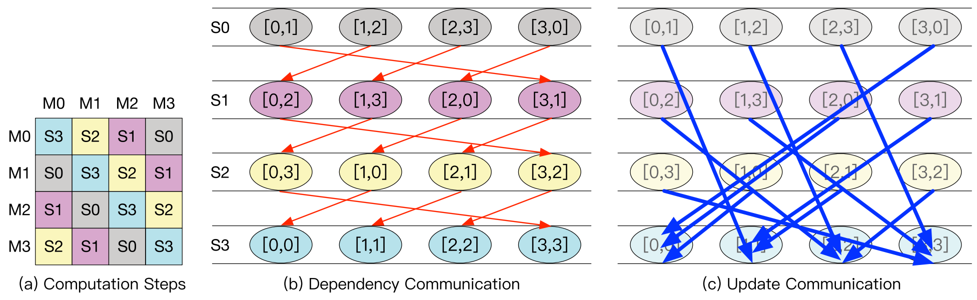

M0 M1 M2 M3
S3 S2 S1 S0
S0 S3 S2 S1
S1 S0 S3 S2
S2 S1 S0 S3
(a) Computation Steps
S0 [0,1] [1,2] [2,3] [3,0]
S1 [0,2] [1,3] [2,0] [3,1]
S2 [0,3] [1,0] [2,1] [3,2]
S3 [0,0] [1,1] [2,2] [3,3]
(b) Dependency Communication
(c) Update Communication

Fig. 9. A Circulant Scheduling Example. Panel (a) shows the graph partitions and their corresponding steps in the schedule. Panels (b, c) show the communication flow.

execution. Without circulant scheduling, a machine has the freedom to process all edges with sources allocated to this machine (a range of rows in Figure 6(a)); with circulant scheduling, during a given step (a part of an iteration), the machine can only process edges in the corresponding subgraph. In another word, the machine loses the freedom to process edges in other steps during this period. The evaluation results in Section 7 will show that the eliminated redundant computation and communication can fully offset the effects of reduced parallelism.

Figure 9 also shows the key difference between dependency and update communication. The dependency communication happens between two steps, because the next step needs to receive it *before* execution to enforce loop-carried dependency. For update communication, each machine will receive from all remote machines by the end of the current iterations, when local reduction and update are performed. The circulant scheduling will not incur much additional synchronization overhead by transferring dependency communication between steps, because it is much smaller than dependency communication. Moreover, before starting a new step, if a machine does not wait for receiving the full dependency communication from the previous step, the correctness is not compromised. With incomplete information, the framework will just miss some opportunities to eliminate unnecessary computation and communication. In fact, Gemini can be considered as a special case without dependency communication.

5.2 Differentiated Dependency Propagation

This section discusses an optimization to further reduce communication. In circulant scheduling, by default, every vertex has dependency communication. For vertices with a lower degree, they have no mirrors on some machines, thus dependency communication is unnecessary. Figure 10 shows the execution of two vertices L and H in basic circulant scheduling. The system has five machines. Two vertices have masters in machine 1. For simplicity, the figure removes the edges for signal functions. The green and red edges are update and dependency messages. For vertex H, every other machine has its mirror. Therefore, the dependency message is propagated across all mirrors and potentially reduces computation and update communication in some mirrors. For vertex L, only machine 2 has its mirror. However, we still propagate its dependency message from machine 1 to machine 5.

One naive solution to avoid unnecessary communication for vertex L is to store the mirror information in each mirror. Before sending the dependency communication of a vertex, we first check the machine number of the next mirror. However, the solution is infeasible for three reasons: First, the memory overhead for storing the information is prohibitive. The space complexity is the same as the total number of mirrors $O(|E|)$. Second, dependency communication becomes complicated in circulant scheduling. Consider a vertex with mirrors in machines 2 and 4, even

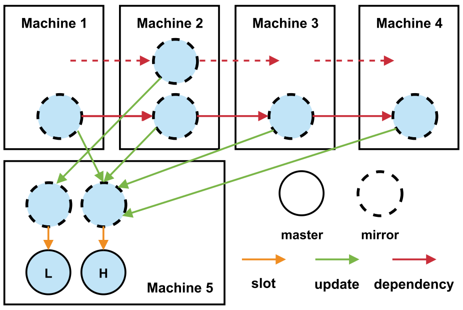


Fig. 10. Differentiated dependency propagation.

when there is no mirror of the vertex on machine 3, we still need to send a message from machine 2 to machine 3, because we cannot discard any message in circulant communication. Third, it does not allow batch communication, since the communication pattern for contiguous vertices are not the same.

To reduce dependency communication with smaller benefits, we propose to differentiate the dependency communication for *high-degree* and *low-degree* vertices. The degree threshold is an empirical constant. The intuition is that dependency communication is the same for the high-degree and low-degree vertices, but the high-degree vertices can *save more update* communication. Therefore, SympleGraph only propagates dependency for high-degree vertices. For low-degree vertices, we can fall back to the original schedule: each mirror directly sends the update messages to the machine with the master vertex.

Differentiated dependency propagation is a trade-off. Falling back for low-degree vertices may reduce the benefits of reducing the number of edges traversed. However, since the low-degree vertices have fewer neighbors, the redundant computation due to loop-carried dependency is also insignificant, because it skips less neighbors. PowerLyra [14] proposed differentiated graph *partition*. One of the key idea is to not replicate low-degree vertices. It reduces the number of mirrors to zero, and the neighbors of the vertices are local. Therefore, it eliminates both the update and dependency communication for these vertices. SympleGraph is orthogonal to graph partition optimizations like PowerLyra. SympleGraph will not change the communication and computation pattern of low-degree vertices, as required by PowerLyra. Instead, SympleGraph enables a new communication for high-degree vertices.

5.3 Hiding Latency with Double Buffering

In circulant scheduling, although disjoint sets of vertices can be executed in parallel within one step, and the computation and update communication can be overlapped, the dependency communication appears in the critical path of execution between steps. Before each step, every machine waits for the dependency message from the predecessor machine. It is not a global synchronization for all machines: synchronization between machine 1 and 3 is independent of that between machine 1 and 2. However, it still impairs performance. Besides the extra latency due to the dependency message, it also incurs load imbalance *within the step*. However, all existing load balancing techniques focus on an *entire iteration* and cannot solve our problem. As a result, the overall performance is affected by the slowest step.

We propose double buffering optimization that enables computation and dependency communication overlap and alleviates load imbalance. Figure 11 demonstrates the key idea with an example. We consider two machines and the first two steps. Specifically, the figure shows the dependency

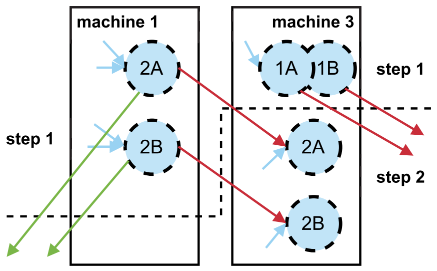


Fig. 11. Double buffering.

communication from machine 1 to machine 3 in step 1 in red. We also add back the blue signal edges to represent the computation on the mirrors. In circulant scheduling, the dependency communication starts after all computation is finished for the mirrors of partition 2 in machine 1.

With double buffering, we divide the mirror vertices in each step into two groups, A and B. First, each machine processes group A and generates its dependency information, which is sent before the processing of vertices in group B. Therefore, the computation on group B is overlapped with the dependency communication for group A, and can be done in the background. In the example, machine 3 will receive the dependency message of group 2A earlier so that the processing of vertices in group 2A in machine 3 does not need to wait until machine 1 finishes processing all vertices in both group 2A and 2B. After the second group is processed, its dependency message is sent, and the current step completes. Before starting the next step, machine 3 only needs to wait for the dependency message for group A, which was initiated earlier before the end of the step.

Double buffering optimization addresses two performance issues. First, at the sender side, group A communication is overlapped with group B, while group B communication can be overlapped with group A computation in the next step. Second, the synchronization wait time is reduced due to reduced load imbalance. Consider the potential scenario of load imbalance in Figure 11, machine 3 (receiver) has much less load in step 1 and proceeds to the next step before machine 1. only waits for the dependency message of group A. Since that message is sent first, it is likely to have already arrived. Without double buffering, machine 3 has to wait for the full completion of machine 1 in step 1.

Importantly, the double buffering optimization can be perfectly combined with the differentiated optimization. We can consider the high-degree and low-degree vertices as two groups. Since processing low-degree vertices does not need synchronization, we can overlap it with dependency communication. In the example, if dependency from machine 1 has not arrived, we can start low-degree vertices in step 2 without waiting.

5.4 Implementation Details

SympleGraph is implemented using C++ based on Gemini and its signal-slot interface. The key component is the dependency communication library.

SympleGraph builds *dependency data structure* from data fields in struct `DepMessage`. Dependency primitives will access the data structures by setting and reading the bits/data of each vertex. For efficient parallel access, we organize the data in **Struct of Arrays (SOA)**. Each data field is instantiated as an array of the type. The size of each array is the number of vertices. The special bit field designed for the control dependency will become a bitmap data structure.

On each machine, SympleGraph starts a dependency communication coordinator thread responsible for transferring dependency message and synchronization. Before execution, coordinator threads set up network connection and initialize the dependency data structures. SympleGraph

```
sendBuf = local Array[DataType];
recvBuf = local Array[DataType];
Bitmap = local Array[bool];
InitBatch();
for (stepId = 0; stepId < cubeNum; stepId++) {
  if (stepId != 0)
    RecvBitMap(); //receive dep. information

  //Next cube for update communication
  upNext = (myId + stepId + 1 + cubeNum) % cubeNum;
  //Next cube for dependency communication
  depNext = (myId - 1 + cubeNum) % cubeNum;

  for (u <- GraphGrid[myId,upNext].vertices) {
    sendBuf[u] = GenUpdate(outNbrs(u));
  }

  if (stepId != (cubeNum-1)) {
    //End of each step except the last
    SendBatch(upNext);
    SendBitMap(depNext);
  } else { //End of last step, local updates
    for (v <- Partition.vertices)
      ApplyUpdate(v, sendBuf(v));
  }

  //At the end of each step (except the first)
  //Master cube handles updates from replicas
  if (stepId != 0) {
    //Batched update from previous step
    RecvBatch();
    for (v <- Partition.vertices)
      ApplyUpdate(v, recvBuf(v));
  }
}
```

(a) GraphS Runtime Implementation

```
1 vertexBuf = local Array[DataType];
2 vertexMapping = local Array[DataType];
3 InitBatch();
4 for (stepId = 0; stepId < cubeNum; stepId++) {
5   curMasterID = (myId + stepId + cubeNum) % cubeNum;
6   recvBuf = &vertexBuf[curMasterID.first_vertexId];
7   sendBuf = recvBuf;
8   for (u <- GraphGrid[myId,curMasterID].vertices) {
9     if(recvBuf[u]==SKIP)
10      break;
11    result,sendBuf[u] = GenUpdateR(recvBuf[u],outNbrs(u));
12    nextCube = vertexMapping[u].get_next();
13    if(result||(nextCube == INT_MAX))
14      addToResultBatch(u,result,nextCube);
15    else
16      addToBatch(u,result,nextCube);
17  }
18  //Sending partial results forward
19  for(i = 1; i <(cubeNum - stepId); i++ ){
20    destId = (myId + i + cubeNum) % cubeNum;
21    SendBatch(destId); //data inserted in Line 16
22  }
23  //Sending final results back to master cube
24  if(stepId!=0){
25    SendResult(curMasterID);
26    //Each step can only receive from one cube
27    recvId = (myId + stepId + cubeNum) % cubeNum;
28    RecvResult(recvId);
29  }
30  //Receive partial results
31  for(i = 1; i < (cubeNum - stepId); i++ ){
32    recvId = (myId + stepId + i + cubeNum) % cubeNum;
33    RecvBatch(i);
34  }
35 }
```

(b) GraphSR Runtime Implementation

Fig. 12. The runtime implementation of GraphS and GraphSR.

also considers NUMA-aware optimizations: We set the affinity of coordinator and the communication buffer for better NUMA locality.

To leverage multi-core hardware in each machine, we start multiple worker threads. During the execution, each worker thread generates the dependency message and notifies the coordinator thread. Before the execution, each worker thread queries the coordinator to check whether the dependency message has arrived. The granularity of worker-coordinator notification is a critical factor for communication latency. If we batch all the communication in worker threads, then the latency of dependency will increase. If we send the message too frequently, then the worker-coordinator synchronization overhead becomes considerable.

To implement circulant scheduling, we change the scheduling order in the framework. For differential propagation, we need to divide the vertices into two groups by their degrees in the preprocessing step. For the degree threshold, we search powers of two with the best performance and use 32 for all evaluation experiments. In the implementation, we generalize double-buffering by supporting more than two buffers to handle different overlap cases. If the processing of low-degree vertices cannot be fully overlapped with dependency communication, then more buffers are necessary.

SympleGraph supports RDMA network using MPI. We use *MPI_Put* for one-sided communication. For synchronization across steps, we use *MPI_Win_lock*/*MPI_Win_unlock* operations to start/end a RDMA epoch on the sender side. It is the "passive target synchronization" where the remote receiver does not participate in the communication. It incurs no CPU overhead on the receiver side.

6 GRAPHS AND GRAPHSR RUNTIME AND ARCHITECTURE

This section discusses the detailed GraphS and GraphSR runtime system implementations. Since both GraphQ and GraphS use circulant scheduling with update communication, we build the

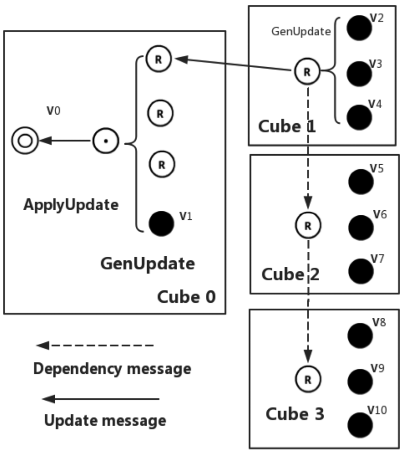


Fig. 13. GraphS runtime.

runtime of GraphS based on GraphQ with two modifications: (1) implementing `GenUpdate` and `ApplyUpdate` APIs, instead of `processEdge`, `reduce`, and `apply`; (2) adding *dependency* communication operations to send and receive Bitmap. Specifically, `SendBitMap` sends BitMap to the next memory cube in circulant scheduling with cube ID as the parameter. `RecvBitMap` receives BitMap from the previous cube in circulant scheduling. In Section 6.1, we describe the basic GraphS runtime in which the master cube of a vertex collects the partial results of `GenUpdate` from all remote replica cubes and generates the final update. It follows the procedure in Figure 5(b). Then, we propose an optimized design GraphSR in which the partial results of `GenUpdate` are passed around cubes, since partial results accumulation can lead to further computation and communication reduction. This optimization is not applicable to all algorithms, we will discuss the observation, applicability, and detailed design in Section 6.2.

6.1 GraphS Runtime System

Figure 12(a) shows the implementation of GraphS runtime. Each cube has (1) `sendBuf` and `recvBuf` for batched inter-cube update communication; (2) `Bitmap` to keep the dependency information dynamically. Batched communication is initialized in all cubes before an iteration starts. An iteration is divided into N steps, where N is the number of cubes. As shown in Figure 9, each cube will have to receive the dependency communication from the previous cube in circulant scheduling except the first step. It is performed by `RecvBitMap` (Line 7). The `Bitmap` of each cube is always written by a remote cube. We calculate destinations of update and dependency communication (Lines 9 ~ 12) according to the patterns in Figures 9(c) and 9(b), respectively. Each cube locally computes the partial results of `GenUpdate` for vertices in the current grid determined by `[myId,upNext]` (Lines 14 ~ 16). These results will be used in calculating the final updates for *destinations* of all edges in the current grid, and are stored in `sendBuf`. In the local `Bitmap`, bits corresponding to the updated vertices are set (`emit_dep` in Figure 7). Next, the partial results and a range of bits in `Bitmap` are sent to the calculated destinations as update and dependency communication, respectively (Lines 20 and 21). The sender cube knows `Bitmap` address in the destination cube, and only non-zero bytes are transferred—if any bit in a byte is non-zero, we transfer the byte. At the end of each step, each cube has to receive the batched update communication and perform `ApplyUpdate` based on the received partial results (`recvBuf`). It is similar to GraphQ, and

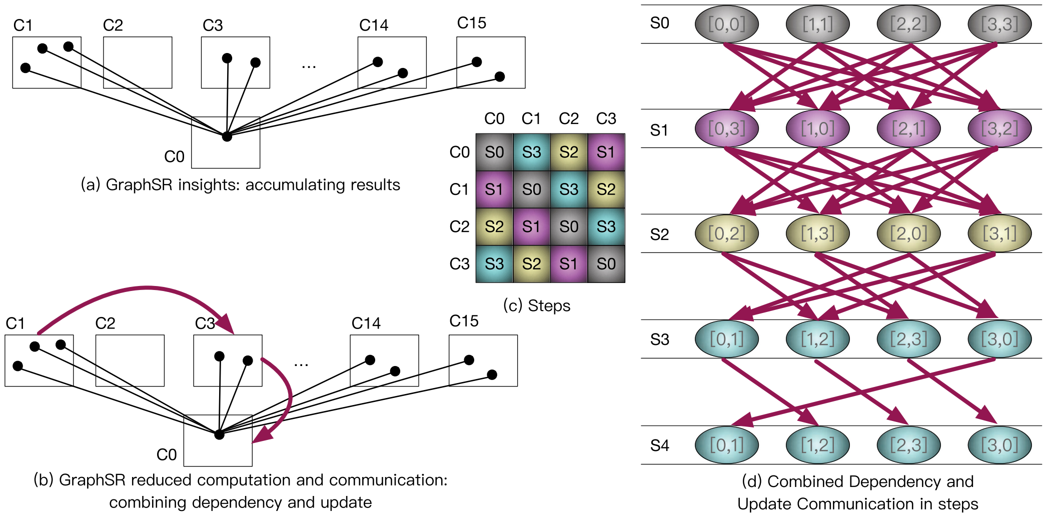

C1
C2
C3
...
C14
C15
C0
(a) GraphSR insights: accumulating results
C0 C1 C2 C3
C0 S0 S3 S2 S1
C1 S1 S0 S3 S2
C2 S2 S1 S0 S3
C3 S3 S2 S1 S0
(c) Steps
C1
C2
C3
...
C14
C15
C0
(b) GraphSR reduced computation and communication: combining dependency and update
S0 [0,0] [1,1] [2,2] [3,3]
S1 [0,3] [1,0] [2,1] [3,2]
S2 [0,2] [1,3] [2,0] [3,1]
S3 [0,1] [1,2] [2,3] [3,0]
S4 [0,1] [1,2] [2,3] [3,0]
(d) Combined Dependency and Update Communication in steps

Fig. 14. GraphSR insights.

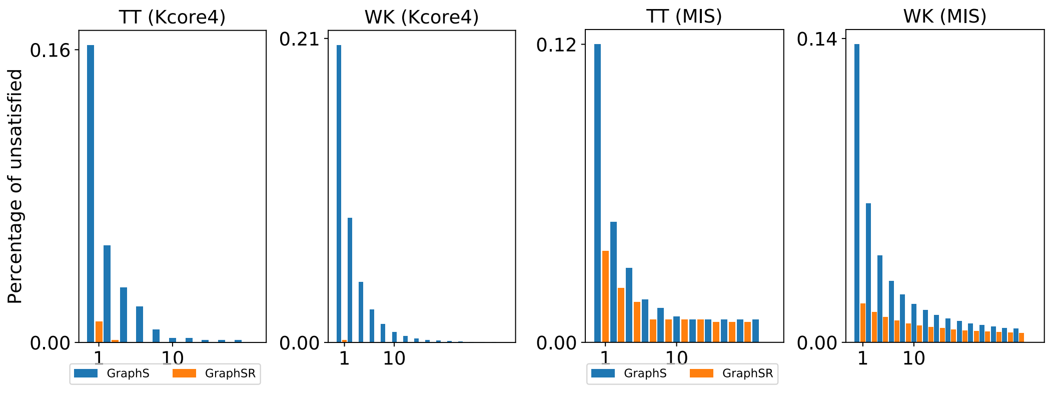
the update communication of the previous step can be overlapped with the computation of the current step. At the end of the last step, all cubes will perform a final and local `ApplyUpdate`. The execution of GraphS runtime with four cubes is illustrated in Figure 13.

6.2 GraphSR Runtime System

Insight. For certain algorithms, it is beneficial to *accumulate* partial results to expedite the termination of correct computation and further reduce redundant work. Consider the k-core algorithm, which finds all vertices with degrees ≥ k, when k = 4, if each cube has less than 4 vertices connected to the master vertex in a remote cube, no computation and communication can be saved. It is shown in Figure 14 (a) with 16 cubes. The black edges are edges of C0. They also indicate the potential update communication. However, it is still different from the exact algorithm semantics, which will stop traversing after 4 neighbors have been identified. Essentially, GraphS runtime only improves GraphQ when 4 or more neighbors are identified in a *single* cube. To truly enforce the precise semantics, partial results have to be *accumulated* and passed around cubes.

Consider a vertex v in the master cube, which first performs local `GenUpdate` (not shown), since if the result[4] for vertex v can be generated locally, there is no need to check the vertex further in other cubes. If the result is not generated, then the master cube will send the current partial results to the next cube in circulant scheduling, which execute `GenUpdate` using *both* its local data and the received partial results. If the "positive" final result (degree ≥ 4) is obtained in a cube, then it will directly send it to the master cube, which updates the master copy of the vertex. Otherwise, the partial result will be sent to the next cube in the fixed circulant scheduling that *has vertices connected to the master vertex v*. This policy avoids unnecessary communication between cubes, which only passes around irrelevant partial results. The ideas are shown in Figure 14(b). The red edges indicate communication messages in GraphSR. Cube 1 generates partial results 3 (no neighbor is found in cube 0), which is passed to cube 3, since cube 2 does not have vertices connected to v in cube 0.

[4]The result in k-core for a vertex is a boolean indicating whether it has ≥ k neighbors.



(a)

(b)

Fig. 15. Percentage of unsatisfied high-degree vertex (Kcore4 & MIS).

Benefits over GraphS. We run a static analysis to simulate the first iteration of K-core (K=4) and MIS algorithm. Figures 15(a) and 15(b) show the percentage of high-degree vertices that are not finished at the end of current step, with graph TT and WK as examples. This percentage is always lower for GraphSR, which leads to less communication.

Implementation. Based on these insights, we can implement GraphSR, a runtime system that further eliminates redundant computation and communication in GraphS. The data movements between cubes in GraphSR are essentially a *combination* of update (for partial results) and dependency communication. Intuitively, communication for a vertex happens when "the condition needs to be checked in the receiver cube with the provided partial result." To compute the first partial result, each cube will execute locally and them follows the circulant scheduling. Thus the step scheduling is circular left shift by one position from Figure 9(a), shown in Figure 14(c). The corresponding execution is shown in Figure 14(d). GraphSR no longer has two types of communications, so we use the same color.

The key feature of GraphSR is that a cube can send multiple batched communications to different remote cubes. It is due to the policy mentioned earlier: for a vertex v, a cube only sends the message to the next cube that has v's neighbors, so certain cube can be skipped, e.g., C_2 in Figure 14(b). The message between two cubes (C_i and C_j) and two consecutive steps includes the batched partial results of vertices that are obtained in C_i and need to be further checked in C_j. Thus, the message between (C_i and C_j) and message between (C_i and C_j) between two given steps contain partial results for *disjoint* set of vertices. As shown in Figure 14(d), with four cubes, the first local cube may send to all three remote cubes (between S_0 and S_1). After the first remote cube, each cube may still send to all three others—when the result may be found and sent back to the master cube (between S_1 and S_2). However, between S_2 and S_3, each cube may only send to two other cubes, this is because the partial results only propagate forward. Eventually, after the last step in the scheduling, each cube can only send one message back to the master cube. In general, with N cubes, the *maximum* number of possible messages between S_i and S_{i+1} is: (N-1) for i=0,1; (N-i) for $i = 2, \ldots, (N-1)$.

For a vertex v, the next cube to send the partial result can be determined statically—it is a function of graph partition. We can obtain that information in pre-processing and store it in a table `vertexMapping` in each cube, each vertex needs $log_2 N$ bits. The overhead of generating this information is negligible, since such information can be directly obtained when assigning each edge to a cube at the graph partitioning time.

The detailed GraphSR runtime implementation is shown in Figure 12(b). At every step, each cube computes curMasterID (Line 5)—the master cube of the incoming partial results. If the condition is satisfied in the current cube or it is the last in the ring, then the result needs to be sent back to cube curMasterID. Each entry in recvBuf has the default value SKIP and it will be only changed if partial result is received, which means that the cube needs to compute the update using received partial result and local vertices. It is done by GenUpdateR (Line 11), which can be generated by adding partial result as an input. If the result is computed (should terminate) or the cube is the last one, then the result needs to be placed in result batch by addToResultBatch (Line 14), which is sent to cube curMasterID (Line 25). If the result is not computed (need to forward), then the partial result is placed in normal batch by addToBatch (Line 16), which is sent to corresponding cube (Line 21). The cubes in the circle receive multiple normal batch at Line 33 and each master cube receives result batch at Line 28. After computing partial results, if a cube does not have anything to send to another one expecting an incoming forward message, it needs to send a small "dummy" message to avoid deadlock. This is not shown in Figure 13.

Applicability. Besides k-core algorithm, the idea of GraphSR also improves MIS, where the key operation is to compare a vertex v's randomly assigned "color" to all its neighbors' color so that all vertices can make consistent decisions on identifying independent sets. We need to check all neighbors of v in different cubes and perform the comparison. The partial result transferred will be whether vertex v's random number if larger (or smaller) than all vertices that have been checked in previous cubes. There are two reasons why GraphSR can also benefit MIS: (1) It computes at local cube in the first step, but GraphS will compute local at last step. The result is affected by graph structure and partitioning, certain data sets are more likely to satisfy the condition at local cube. (2) GraphS only sends message for high-degree vertices (see Section 5.2, but GraphSR propagates dependence for all vertexes with combined update and dependency communication.

GraphSR cannot benefit algorithms that do not have the above property. For example, in bottom-up BFS, the condition is that any neighbor belongs to the current frontier. In the beginning and end of the execution, there are few vertices in the frontier, so many checks turn out to be false. In GraphS, these cases do not generate update communication to master cube, while GraphSR still needs to circulate the partial results across cubes. Kmeans is similar, since it can be considered as BFS with multiple roots.

The key reason why SympleGraph does not implement the method used in GraphSR is that inter-machine communication is far less efficient than that of PIM. The partial result propagation requires a relatively small communication batch size. If the batch size is too large, then the partial result may not be delivered to its next node in circular scheduling on time. As a result, the node may wait for the partial results. However, small communication batch size is not acceptable in the inter-machine network, since the latency is too high and small batch size cannot efficiently amortize the overhead of latency. Therefore, implementing the partial results propagation on the distributed environment reduces the network bandwidth utilization and leads to worse performance.

7 SYMPLEGRAPH EVALUATION

We evaluate SympleGraph, Gemini [85] and D-Galois [18]. D-Galois is a recent state-of-the-art distributed graph processing frameworks with better performance than Gemini with 128 to 256 machines.

In the following, we describe the evaluation methodology. After that, we show the results of several important aspects: (1) comparison of overall performance among the three frameworks; (2) reduction in communication volume and computation cost; (3) scalability; and (4) piecewise contribution of each optimization.

Table 1. Graph Datasets

Graph	Abbrev.	|V|	|E|	$\frac{\|V'\|}{\|V\|}$
Twitter-2010 [38]	tw	42M	1.5 B	0.13
Friendster [41]	fr	66M	1.8 B	0.31
R-MAT-Scale27-E32	s27	134M	4.3 B	0.12
R-MAT-Scale28-E16	s28	268M	4.3 B	0.09
R-MAT-Scale29-E8	s29	537M	4.3 B	0.04
Clueweb-12 [11, 59]	cl	978M	43 B	0.12
Gsh-2015 [10]	gsh	988M	34 B	0.28

|V′| is the number of high-degree vertices.

7.1 Evaluation Methodology

System configuration. We use three clusters in the evaluation: (1) Cluster-A is a private cluster with 16 nodes. In each node, there are 2 Intel Xeon E5-2630 CPUs (8 cores/CPU) and 64 GB DRAM. The operating system is CentOS 7.4. MPI library is OpenMPI 3.0.1. The network is Mellanox InfiniBand FDR (56 Gb/s). The following evaluation results are conducted in Cluster-A unless otherwise stated. (2) Cluster-B is Stampede2 Skylake (SKX) at the Texas Advanced Computing Center [55]. Each node has 2 Intel Xeon Platinum 8160 (24 cores/CPU) and 192 GB DRAM with 100 Gb/s interconnect. It is used to reproduce D-Galois results, which requires 128 machines and fails to fit in Cluster-A. (3) Cluster-C consists of 10 nodes. Each node is equipped with two Intel Xeon E5-2680v4 CPUs (14 cores/CPU) and 256 GB memory. The network is InfiniBand FDR (56 Gb/s). It is used to run the two large real-world graphs (Clueweb-12 and Gsh-2015), which requires larger memory and fails to fit in Cluster-A.

Graph Dataset. The datasets are shown in Table 1. There are four real-world datasets and three synthesized scale free graphs with R-MAT generator [13]. We use the same generator parameters as in Graph500 benchmark [27].

For experiments in Cluster-A, we generate three largest possible synthesized graph that fits in its memory. Any larger graph will cause an out-of-memory error. The scales (logarithm of the number of vertices) are 27, 28, and 29 and the edge factor (average degree of a vertex) are 32, 16, and 8, respectively. To run undirected algorithms using directed graphs, we consider every directed edge as its undirected counterpart. To run directed algorithms using undirected graphs, we convert the undirected datasets to directed graphs by adding reverse edges.

Graph Algorithms. We evaluate five algorithms discussed before. We use the reference implementations when they are available in Gemini and D-Galois. While SympleGraph only benefits the bottom-up BFS, we use adaptive direction-switch BFS [68] that chooses from both top-down and bottom-up algorithms in each iteration.[5, 6] We follow the optimization instructions in D-Galois by running all partition strategies provided and report the best one as the baseline.[7]

For BFS, we average the experiment results of 64 randomly generated non-isolated roots. For each root, we run the algorithm five times. For K-core, 2-core is a subroutine widely used in strongly connected component [33] algorithm. We also evaluate other values of K. For K-means,

[5]Adaptive switch is not available in D-Galois. For fair comparison, we implement the same switch in D-Galois.
[6]Graph sampling implementation is not available in D-Galois.
[7]We exclude Jagged Cyclic Vertex-Cut and Jagged Blocked Vertex-Cut (in all algorithms) and Over decomposed by 2/4 Cartesian Vertex-Cut (in K-core), because the reference implementations either crashed or produced incorrect results.

Table 2. K-core Runtime (in Seconds) (Cluster-A)

Graph	K	Gemini	SympleG.	Speedup
tw	4	1.9663	1.3009	1.51
	8	2.9752	2.0595	1.44
	16	4.9062	3.2957	1.49
	32	5.8374	3.7916	1.54
	64	7.5694	5.1717	1.46
fr	4	14.7322	10.3543	1.42
	8	10.1319	6.7909	1.49
	16	11.5904	7.4135	1.56
	32	21.8317	13.4914	1.62
	64	17.9096	11.4387	1.57

we choose the number of clusters as $\sqrt{|V|}$ and runs the algorithm for 20 iterations. For algorithms other than BFS, we run the application 20 times and average the results.

7.2 Performance

Table 4 shows the execution time of all systems. SympleGraph outperforms both Gemini and D-Galois with 1.46× geomean (up to 3.05×) speedup over the best of the two. For the three synthesized graphs with the same number of edges but different edge factor (s27, s28, and s29), graphs with larger edge factor have slightly higher speedup in SympleGraph. For K-core, the numbers in parenthesis use the optimal algorithm with linear complexity in the number of nodes and no loop dependency [47]. It is slower than SympleGraph for large synthesized graphs, but significantly faster for Twitter-2010 and Friendster. The reason is that the algorithm is suitable for graphs with large diameters. Although real-world social graphs have relatively small diameters, they usually have a long link structure attached to the small-diameter core component.

K-core. Table 2 shows the execution time (using 8 Cluster-A nodes) for different values of K. SympleGraph has consistent speedup over Gemini regardless of K.

Large Graphs. We run Gemini and SympleGraph with the two large real-world graphs (Gsh-2015 and Clueweb-12) on Cluster-C. SympleGraph has no improvement for BFS and K-means in Clueweb-12. The reason is that bottom-up algorithm efficiency depends on graph property. In cl, it is slower than top-down BFS for most iterations, so they are not chosen by the adaptive switch. In other test cases, SympleGraph is noticeably better than Gemini.

7.3 Computation and Communication Reduction

The source of performance speedup in SympleGraph is mainly due to eliminating unnecessary computation and communication with precisely enforcing loop-carried dependency. In graph processing, the number of edges traversed is the most significant part of computation. Table 5 shows the number of edges traversed in Gemini and SympleGraph. The first two columns are edge traversed in Gemini and SympleGraph. The last column is their ratio. We see that SympleGraph reduces edge traversal across all graph datasets and all algorithms with a 66.91% reduction on average.

For communication, Gemini and other existing frameworks only have update communication, while SympleGraph reduces updates but introduces dependency communication. Table 6 shows the breakdown of communication in SympleGraph. Communication size is counted by message size in bytes and all the numbers are normalized to the total communication in Gemini. The first

Table 3. Execution Time (in Seconds) on Large Graphs (Cluster-C)

Graph	App.	Gemini	SympleG.	Speedup
gsh	BFS	4.5843	4.6031	1.00
	MIS	7.3186	4.1530	1.76
	K-core	24.1753	13.4465	1.80
	K-means	84.7207	75.7227	1.12
	Sampling	4.6578	3.4686	1.34
cl	BFS	16.8839	17.9272	1.00
	MIS	11.9406	6.8330	1.75
	K-core	171.8570	97.7020	1.76
	K-means	128.5634	142.6216	1.00
	Sampling	4.5093	3.6143	1.25

Table 4. Execution Time (in Seconds) (Cluster-A)

	Graph	Gemini	D-Galois	SymG.	Speedup
BFS	tw	0.608	2.053	**0.264**	2.30
	fr	1.212	4.993	**0.706**	1.72
	s27	1.054	2.681	**0.733**	1.44
	s28	1.325	3.682	**0.976**	1.36
	s29	1.760	5.356	**1.372**	1.28
K-core	tw	3.021(0.184)	4.125	**2.190**	1.38
	fr	11.258(0.580)	17.213	**7.390**	1.52
	s27	2.754(1.885)	3.512	**1.640**	1.68
	s28	4.432(4.779)	6.056	**2.663**	1.66
	s29	5.413(10.330)	8.534	**3.806**	1.42
MIS	tw	2.081	4.056	**1.421**	1.46
	fr	2.363	5.045	**1.754**	1.35
	s27	2.720	5.329	**1.861**	1.46
	s28	3.031	7.110	**2.408**	1.26
	s29	3.600	8.620	**2.835**	1.27
K-means	tw	17.590	56.748	**12.688**	1.39
	fr	19.212	78.526	**13.143**	1.46
	s27	27.626	61.598	**19.279**	1.43
	s28	34.393	86.632	**26.919**	1.28
	s29	52.087	116.307	**41.760**	1.25
Sampling	tw	**0.786**		0.867	*0.91*
	fr	1.180		**0.977**	1.21
	s27	1.388	N/A	**1.090**	1.27
	s28	2.051		**1.331**	1.54
	s29	2.932		**1.869**	1.57

(SympleGraph.upt) and second (SympleGraph.dep) column show update and dependency communication, respectively. The last column is the total communication of SympleGraph.

There are two important observations. First, s27, s28, and s29 have the same total number of edges, while s27 traverses consistently less edges than s28 and s29 in all algorithms. On average, SympleGraph on s27 traverses 24.8% edges compared with Gemini, while on s29 traverses 32.8%.

Table 5. Number of Traversed Edges (Normalized to Total Number of Edges in the Graph) (Cluster-A)

	Graph	Gemini	SympG.	SympG./Gemini
BFS	tw	0.4383	0.2214	0.5051
	fr	0.8537	0.3435	0.4024
	s27	0.3089	0.0870	0.2815
	s28	0.3586	0.1348	0.3760
	s29	0.4716	0.1879	0.3985
K-core	tw	2.6421	1.1986	0.4537
	fr	11.3283	3.1951	0.2820
	s27	1.1188	0.3498	0.3126
	s28	1.8717	0.6165	0.3294
	s29	2.4237	1.0513	0.4338
MIS	tw	3.9014	1.9750	0.5062
	fr	5.4431	2.0479	0.3762
	s27	3.1328	0.8717	0.2782
	s28	3.4390	1.0174	0.2958
	s29	3.7762	1.1970	0.3170
K-means	tw	13.3972	5.5608	0.4151
	fr	2.5798	1.8989	0.7361
	s27	5.6167	1.7196	0.3062
	s28	8.8354	2.7847	0.3152
	s29	13.6472	5.3375	0.3911
Sampling	tw	1.0313	0.2143	0.2078
	fr	1.2097	0.1290	0.1066
	s27	1.1096	0.0709	0.0639
	s28	1.1498	0.0966	0.0840
	s29	1.1912	0.1172	0.0984

When the graph structure is similar (R-MAT), the number of traversed edges is less in graphs with a larger average degree. A large average degree means more high-degree vertices that SympleGraph optimizes in differentiated computation. Therefore, s27 has more potential edges when considering reducing computation. Second, in terms of total communication size, SympleGraph is less than Gemini in all algorithms except graph vertex sampling. For these algorithms, control dependency communication is one bit per vertex, because the dependency information indicates whether the vertex in the previous step has skipped the loop. For graph sampling, data dependency communication is the current prefix sum. It is one floating-point number for one vertex; thus total communication might increase.

7.4 Scalability

We first compare the scalability results of SympleGraph with Gemini and D-Galois, running MIS on graph s27 (Figure 16). The execution time is normalized to SympleGraph with 16 machines. The data points for Gemini and SympleGraph with 1 machine are missing, because the system is out of memory. Both Gemini and SympleGraph achieves the best performance with 8 machines. D-Galois scales to 16 machines, but its *best performance requires 128 to 256 machines* according to Reference [18]. In summary, SympleGraph is consistently better than Gemini and D-Galois with 16 machines.

Table 6. SympleGraph Communication Breakdown (Normalized to Total Communication Volume in Gemini) (Cluster-A)

	Graph	SymG.upt	SymG.dep	SymG
BFS	tw	0.7553	0.0446	0.7999
	fr	0.4657	0.0429	0.5085
	s27	0.4151	0.0175	0.4326
	s28	0.4855	0.0193	0.5047
	s29	0.5993	0.0154	0.6147
K-core	tw	0.5377	0.0074	0.5450
	fr	0.3646	0.0074	0.3719
	s27	0.3705	0.0051	0.3755
	s28	0.3987	0.0051	0.4038
	s29	0.5028	0.0039	0.5067
MIS	tw	0.4721	0.0313	0.5034
	fr	0.3639	0.0259	0.3898
	s27	0.3053	0.0199	0.3252
	s28	0.3336	0.0208	0.3544
	s29	0.4127	0.0160	0.4287
K-means	tw	0.6854	0.0250	0.7103
	fr	0.7044	0.0393	0.7437
	s27	0.3306	0.0100	0.3406
	s28	0.3797	0.0118	0.3915
	s29	0.5188	0.0106	0.5294
Sampling	tw	0.1877	1.1578	1.3455
	fr	0.1637	0.7238	0.8875
	s27	0.1706	0.6558	0.8264
	s28	0.2106	0.7050	0.9157
	s29	0.2565	0.7504	1.0069

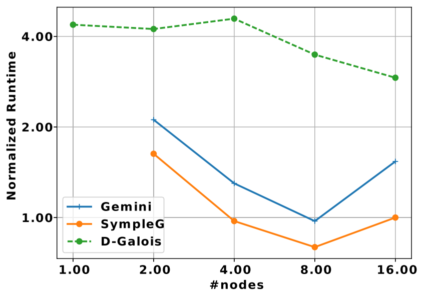


Fig. 16. Scalability (MIS/s27).

From 8 to 16 machines, SympleGraph has a smaller slowdown compared with Gemini, thanks to the reduction in communication and computation. Thus, SympleGraph scales better than Gemini.

COST. The COST metric [50] is an important measure of scalability for distributed systems. It is the number of cores a distributed system need to outperform the fastest single-thread implementation. We use the MIS algorithm in Galois [54] and s27 graph as the single-thread baseline.

Table 7. Execution Time (in Seconds) of MIS Using the Best-performing Number of Nodes (in Parenthesis) on Cluster-B

Graph	D-Galois	SympleGraph
tw	1.321(128)	1.113(2)
fr	1.355(128)	0.823(4)
s27	1.258(128)	0.911(4)
s28	1.380(128)	1.159(4)
s29	1.565(128)	1.420(4)

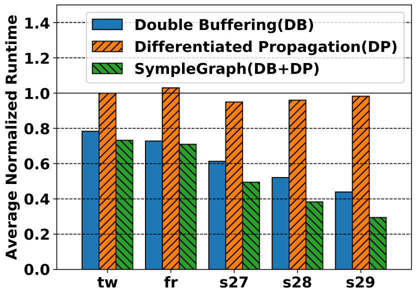


Fig. 17. Analysis of optimizations (baseline is SympleGraph with only circulant scheduling).

The COST of Gemini and SympleGraph is 4, while the COST of D-Galois is 64. We also use the BFS algorithm in GAPBS [8] and tw graph as another baseline. GAPBS finishes in 2.29 s, while SympleGraph takes 2.66 and 1.83 s for 2 and 3 threads, respectively. The cost of SympleGraph is 3.

D-Galois. To evaluate the best performance of D-Galois, We reproduce the results with Cluster-B. The results are shown in Table 7. As the SKX nodes have more powerful CPUs and network, SympleGraph requires less number of nodes (2 or 4 nodes) for the best performance. D-Galois achieves similar or worse performance with 128 nodes. While D-Galois scales better with a large number of nodes, running increasingly common graph analytics applications in the supercomputer is *not* convenient. In fact, for these experiments, the jobs have waited for days to be executed. Based on the results, SympleGraph on a local cluster with 4 nodes can fulfill the work of D-Galois with 128 nodes. We believe using SympleGraph on a small-scale distributed cluster is the most convenient and practical solution.

7.5 Analysis of SympleGraph Optimizations

In this section, we analyze the piecewise contribution of the proposed optimizations over circulant scheduling, i.e., differential dependency propagation, and double buffering. We run all applications on four versions of SympleGraph with different optimizations enabled. Due to space limit, Figure 17 only shows the geometric average results of all algorithms. For each graph dataset, we normalize the runtime to the version with basic circulant scheduling. Note that here the baseline is not Gemini.

Double buffering effectively reduce the execution time in all cases. It successfully hides the latency of dependency communication and reduces synchronization overhead. Differential propagation optimization alone has little performance impact, because synchronization is still the bottleneck without double buffering. When combined with double buffering, differential propagation has a noticeable effect. This shows that our trade-off consideration in update and dependency

Table 8. Datasets

Graph	#Vertices	#Edges
ego-Twitter (TT) [48]	81K	2.4M
Enwiki2013 (WK) [77]	4.2M	101M
LiveJournal (LJ) [42]	4.8M	69M
Twitter2010 (TW) [38]	42M	1.5B
Friendster (FR) [41]	66M	1.8B

communication is effective. Overall, when all optimizations are applied, the performance is always better than individual optimization.

8 GRAPHS AND GRAPHSR EVALUATION

8.1 Evaluation Methodology

PIM System configuration. We evaluate GraphS based on zSim [63], a scalable x86-64 multicore simulator. We modified zSim according to HMC's memory and interconnection model, heterogeneous compute units, on-chip network and other hardware features. While zSim does not natively support HMC interconnection simulation, we insert a NOC layer between LLC and memory to simulate different intra-cube and inter-cube memory bandwidth. The results are validated against NDP [22]. For compute units, we use 256 single-issue in-order cores in all test cases. Each core has 32 KB L1 instruction cache and 64K L1 data cache. Cache line size is 64 B and simulation frequency is 1,000 MHz. Each core has a 16-entry message queue and 32 KB L1 instruction cache, with no L2 or shared cache. For memory configuration, we use 16 cubes (8 GB capacity, 512 banks). The cubes are connected with the Dragonfly topology [37]. The maximal internal data bandwidth of each cube is 320 GB/s. For meaningful comparison, the configurations are the same as GraphQ [86]

The energy consumption of the inter-cube interconnect is estimated as two components: (a) dynamic energy, which is proportional to the number of flit transfer events that happen among each pair of cubes; (b) static energy, which corresponds to the energy cost when the interconnect is plugged in power but in idle state (i.e., no transfer event happens). We use zSim to count the number of transfer events and use ORION 3.0 [36] to model the dynamic and static power of each router. We calculate the leakage energy of the whole interconnect from the flit transfers. We also validated Table 1 in Reference [72] with McPAT [43].

Graph Dataset. We tested with TT,WK,LJ,TW,FR in Table 8. These data sets are smaller compared to the evaluation on real cluster due to the simulation speed limitation. To run algorithm on directed graphs, we convert the un-directed datasets to directed graphs by adding reverse edges. For large graphs (TW, FR), due to the slow simulation speed, we show results of the first several iterations and the speedups are not used in calculating average speedups.

8.2 Performance Improvements

Figure 18 shows the execution cycles of GraphS and GraphSR normalized to GraphQ [86]. For BFS and Kmeans, speedup is on average 1.57× and 1.86× and at maximum 1.70× and 2.43× . For kcore-{2,3,4}, GraphS's speedup is on average 2.96×, 2.93×, 2.84× and 4.37×, 4.09×, 3.86× at maximum, and GraphSR's speedup is on average 13.98×, 13.92×, 12.52× and 19.91×, 17.43×, 15.83× at maximum. When K becomes lager, the speedup gradually becomes less. For MIS, since the second phase is not optimized, the speedup for GraphS is on average 1.60× and at maximum 1.89× and for GraphSR on average 9.67× and at maximum 14.31× (slightly less than that of kcore). GraphS outperforms baseline with a speedup up to 2.30× (1.55× on average).

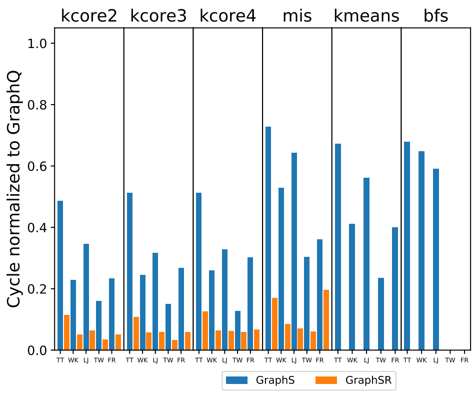


Fig. 18. GraphS and GraphSR runtime.

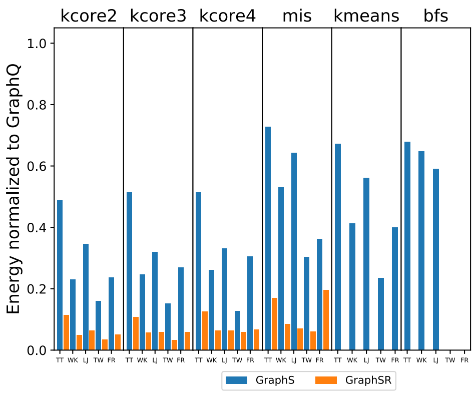


Fig. 19. GraphS and GraphSR energy.

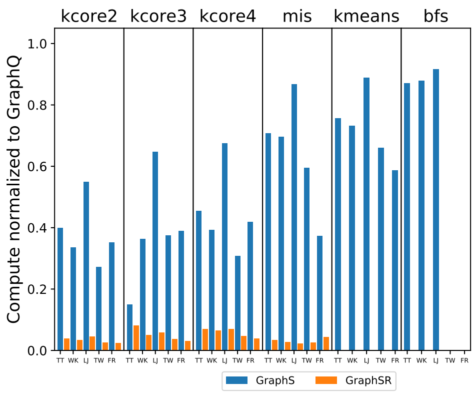


Fig. 20. GraphS and GraphSR actual compute.

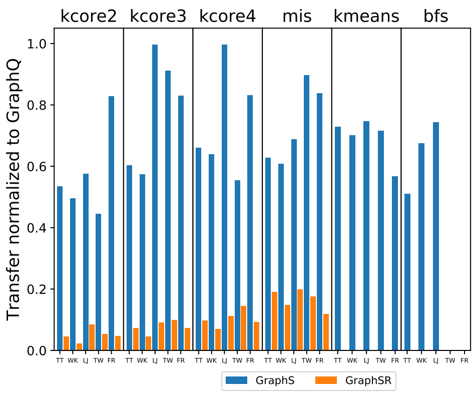


Fig. 21. GraphS and GraphSR communication.

8.3 Breakdown and Load Imbalance

The Figure 22 shows the breakdown of runtime into communication, computation and synchronization. For GraphSR,*Send* indicates the runtime percentage of sending accumulated results. For GraphS and GraphQ, Communication time was overlapped an included in the Sync time. The load imbalance of GraphS with dependancy message is slightly better than GraphQ in most cases. This is because vertexes with higher degrees are more likely to satisfy dependency, removing unnecessary computation with dependency message can help to reduce the imbalance. GraphSR cuts down more than 80% of the total runtime, and lead to very imbalance result. This imbalance is due to graph partition and nature of this method. Given the significant speedups of GraphSR, we do not believe it is a critical issue.

8.4 Energy Reduction

The Figure 19 shows the interconnect energy consumption of GraphS compared with GraphQ. The energy cost of the interconnect consists of both the static consumption and the dynamic consumption, which are determined by the execution time (performance) and communication amount, respectively. GraphS reduces energy cost by 51.6% on average and 76.9% at maximum. GraphSR reduces energy cost by 91.41% on average and 94.99% at maximum.

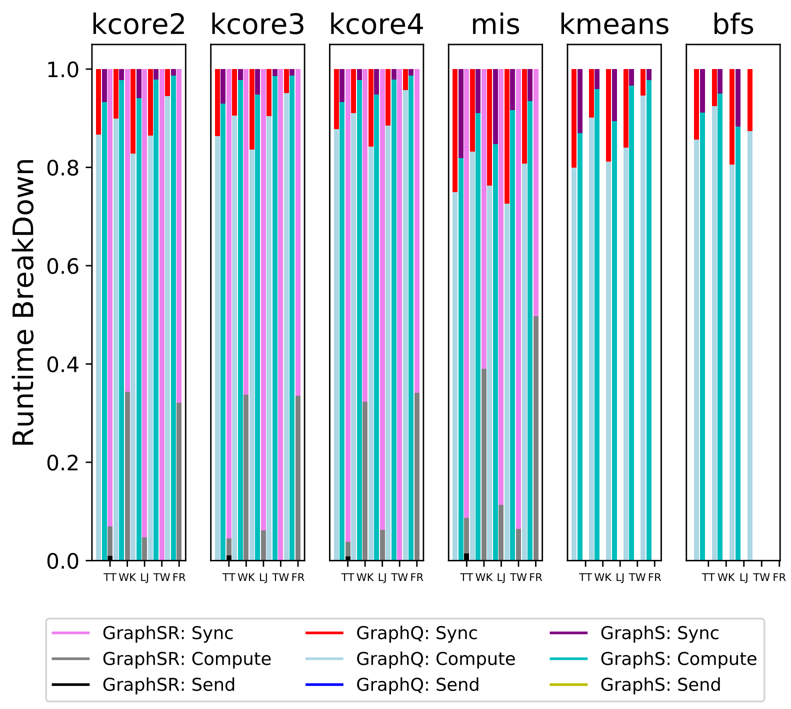


Fig. 22. Runtime breakdown of GraphSR, GraphQ, and GraphS.

8.5 Computation and Communication Reduction

We summed the number of edges computed and number of apply to estimate the computation cost, as shown in The Figure 20. GraphSR can reduce the actual computation to only 4.91% on average and 2.18% at maximum. GraphS can reduce to 62.6% on average and 14.9% at maximum.

The Figure 21 shows the reduction of the communication, GraphSR reduces a huge proportion of communication. For example, in K-core algorithm (K = 2), for vertices with 2 or more edges at first cube, there would be no communication, for vertices with 1 edge at first cube, there would be only one message needed, because if it only has one edge, then only a result be sent to master, if it has more edges at the next cube, then just partial result send to the next cube and it would satisfy there. For GraphS, in dataset like liveJournal[42], as K grows larger, there is little save of communication. GraphS could reduces transfer to 67.2% on average and 49.5% at maximum. And GraphSR could reduces transfer to 9.77% on average and 2.21% at maximum.

In distributed cluster, we calculated the number of edges traversed to show computation reduction. As in Table 5, GraphS reduces edge traversed to 43.6% on average and 28.2% at maximum. For communication, the *upt msg* and *dep msg* column in Table 6 indicate the amount of update and dependency communication, and the *total comm* column means the total communication. All normalized to total communication in Gemini. We can see that GraphS indeed incurs much less total communication than Gemini.

9 RELATED WORK

BFS Systems. References [9, 12] are distributed BFS systems for high performance computing. They enforce loop-carried dependency only for BFS and a specific graph partition. SympleGraph works for general graph algorithms and data partitions.

Edge-centric Graph Systems. X-stream [61] proposes edge-centric programming model. It is motivated by the fact that sequential access bandwidth is larger than random bandwidth for all storage (memory and disk). X-stream partitions the graph into edge blocks and process all the edges in the block sequentially. However, the updates to the destination vertices are random. To avoid random access, X-stream maintains an update list and append the updates sequentially. For each vertex, its updates are scattered in the list. It is infeasible to track the dependency and skip computation in X-stream. Edge-centric systems have other drawbacks and recent state-of-the-art systems are vertex-centric. Therefore, SympleGraph is based on vertex-centric programming model.

Asynchronous Graph Systems. References [46, 73–75] propose to relax the dependency of different vertex functions H across iterations. SympleGraph enforces dependency in I (in Definition 2.1) within one iteration. The dependency is different and thus the optimizations are orthogonal. We will leave it as future work to enable both in one system.

Graph compiler. IrGL [57] and Abelian [23] are similar to the first analysis part in SympleGraph. IrGL focuses on intermediate representation and architecture-specific (GPU) optimizations. Abelian automates some general communication optimizations with static code instrumentation. For example, on-demand optimization reduces communication by recording the updates and sending only the updated values. SympleGraph also uses instrumentation, but the objective is to transform loop-carried dependency, which is not explored in graph compilers.

Graph Domain Specific Language. Many graph DSLs leverage algorithm information by asking the users to program in a new programming interface to express new semantics. GraphIt [84] and GreenMarl [31] are designed for shared-memory graph processing. SocialLite [65] and GRAPE [20] are DSLs for distributed processing. However, they did not address the dependency issue described in this article. For example, GRAPE describes graph algorithms with "partial evaluation," "incremental evaluation" and "combine." Its system implementation is not efficient: the reported distributed performance on 24 machines is worse than single-thread naive implementation on a laptop [49].

Graph Processing Architectures. Tesseract [1] is the first PIM-based accelerator and is the baseline of this article. [56] is an accelerator that support dependence tracking in asynchronous graph processing. GraphPIM [53] demonstrates the performance benefits for graph applications by adding the atomic operations to PIM. Graphicionado [30] is a high performance customized graph accelerator, based on specialized memory subsystem, instead of PIM. GraphP [83] proposes a graph partitioning method that reduces inter-cube communication. GraphQ [86] further reduces communication at intra-cube, inter-cube, inter-node levels. [5] characterized the memory system performance of graph processing workloads and proposed a physically decoupled prefetcher that improves the performance of these workloads. [35, 52] explores online traversal scheduling strategies that exploit the community structure of real-world graphs to improve locality. GraphS is a synchronous graph processing accelerator. It eliminates both communication and computation in PIM.

10 CONCLUSION

This article proposes novel graph processing frameworks for distributed system and Processing-In-Memory architecture that precisely enforces loop-carried dependency, i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. Our approach instruments the UDFs to express the loop-carried dependency, then the distributed execution framework enforces the precise semantics by performing dependency propagation dynamically. Enforcing loop-carried dependency requires the sequential processing of the neighbors of each vertex distributed in different nodes—machines or memory cubes. We propose to use circulant scheduling in the framework to allow different nodes to process disjoint sets of edges/vertices in parallel while satisfying the sequential requirement. Moreover, the precise loop-carried dependency can be selectively applied only to the large-degree vertices, reducing the additional dependency propagation communication that does not exist before. Combined, they achieve an excellent trade-off between precise semantics and parallelism—the benefits of eliminating total unnecessary computation and communication offset the reduced parallelism. We implemented a new distributed graph processing framework SympleGraph, and two variants of runtime systems—GraphS and GraphSR—for PIM-based graph processing architecture. In a 16-node cluster, SympleGraph outperforms Gemini and D-Galois on average by 1.42× and 3.30×, and up to 2.30× and 7.76×, respectively. The communication reduction compared to Gemini is 40.95% on average and up to 67.48%. Compared to GraphQ, the state-of-the-art PIM-based graph processing architecture, GraphS achieves on average 2.2× (maximum 4.37×) speedup, on average 32.8% (maximum 50.5%) inter-cube communication reduction. They lead to 51.6% energy saving on average. With partial results propagation, GraphSR achieves on average 12.5× (maximum 19.91×) speedup, on average 90.23% (maximum 97.79%) inter-cube communication reduction. They lead to 91.4% energy saving on average.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, 105–117.

[2] Tero Aittokallio and Benno Schwikowski. 2006. Graph-based methods for analysing networks in cell biology. *Brief. Bioinform.* 7, 3 (2006), 243–255.

[3] Andrei Alexandrescu and Katrin Kirchhoff. 2007. Data-driven graph construction for semi-supervised graph-based learning in NLP. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL'07)*. 204–211.

[4] ARM. 2009. ARM Cortex-A5 Processor. Retrieved from http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

[5] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, 373–386.

[6] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner et al. 2018. Relational inductive biases, deep learning, and graph networks. Retrieved from https://arXiv:1806.01261.

[7] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 12, 10 pages. Retrieved from http://dl.acm.org/citation.cfm?id=2388996.2389013.

[8] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. Retrieved from https://arXiv:cs.DC/1508.03619.

[9] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Proceeding sof the IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*. IEEE, 1618–1627.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*. ACM, 587–596.

[11] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*. ACM, 595–602.

[12] Aydin Buluc, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patterson. 2017. Distributed-memory breadth-first search on massive graphs. Retrieved from https://arXiv:1705.04590.

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 442–446.

[14] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[15] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. 2004. Graph-based technologies for intelligence analysis. *Commun. ACM* 47, 3 (Mar. 2004), 45–47. https://doi.org/10.1145/971617.971643

[16] Hybrid Memory Cube Consortium. 2015. *Hybrid Memory Cube Specification Version 2.1*. Technical Report.

[17] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 34, 4 (2018), 640–653.

[18] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 752–768. https://doi.org/10.1145/3192366.3192404

[19] Anton J. Enright and Christos A. Ouzounis. 2001. BioLayout—An automatic graph layout algorithm for similarity visualization. *Bioinformatics* 17, 9 (2001), 853–854.

[20] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing sequential graph computations. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 495–510. https://doi.org/10.1145/3035918.3035942

[21] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Trans. Knowl. Data Eng.* 19, 3 (2007), 355–369.

[22] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE, 113–124.

[23] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. 2018. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Proceedings of the European Conference on Parallel Processing*. Springer, 249–264.

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 17–30. Retrieved from http://dl.acm.org/citation.cfm?id=2387880.2387883.

[25] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 599–613. Retrieved from http://dl.acm.org/citation.cfm?id=2685048.2685096.

[26] Amit Goyal, Hal Daumé III, and Raul Guerra. 2012. Fast large-scale approximate graph construction for nlp. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, 1069–1080.

[27] Graph500. 2010. Graph 500 Benchmarks. Retrieved from http://www.graph500.org.

[28] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, New York, NY, 855–864. https://doi.org/10.1145/2939672.2939754

[29] Ziyu Guan, Jiajun Bu, Qiaozhu Mei, Chun Chen, and Can Wang. 2009. Personalized tag recommendation using graph-based ranking on multi-type interrelated objects. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 540–547.

[30] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.

[31] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. *SIGPLAN Not.* 47, 4 (Mar. 2012), 349–362. https://doi.org/10.1145/2248487.2151013

[32] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis (SC'15). ACM, New York, NY, Article 58, 12 pages. https://doi.org/10.1145/2807591.2807620

[33] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY, Article 92, 11 pages. https://doi.org/10.1145/2503210.2503246

[34] Imranul Hoque and Indranil Gupta. 2013. LFGraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*. ACM, New York, NY, Article 9, 17 pages. https://doi.org/10.1145/2524211.2524218

[35] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2016. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro* 36, 3 (2016), 105–117. https://doi.org/10.1109/MM.2016.12

[36] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. 2012. ORION 2.0: A power-area simulator for interconnection networks. *IEEE Trans. Very Large Scale Integr. Syst.* 20, 1 (Jan. 2012), 191–196. https://doi.org/10.1109/TVLSI.2010.2091686

[37] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, 145–156.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, 591–600. https://doi.org/10.1145/1772690.1772751

[39] Nicolas Le Novere, Michael Hucka, Huaiyu Mi, Stuart Moodie, Falk Schreiber, Anatoly Sorokin, Emek Demir, Katja Wegner, Mirit I. Aladjem, Sarala M. Wimalaratne, et al. 2009. The systems biology graphical notation. *Nature Biotechnology* 27, 8 (2009), 735–741.

[40] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, et al. 2014. 25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'14)*. IEEE, 432–433.

[41] Jure Leskovec and Andrej Krevl. 2014. friendster. Retrieved from https://snap.stanford.edu/data/com-Friendster.html.

[42] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.* 6, 1 (2009), 29–123.

[43] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 469–480.

[44] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (Apr. 2012), 716–727. https://doi.org/10.14778/2212351.2212354

[45] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, 135–146. https://doi.org/10.1145/1807167.1807184

[46] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys'19)*. ACM, New York, NY, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[47] David W. Matula and Leland L. Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM* 30, 3 (1983), 417–427.

[48] Julian McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates, 539–547. Retrievedfromhttp://dl.acm.org/citation.cfm?id=2999134.2999195.

[49] Frank McSherry. 2017. COST in the land of databases. Retrieved from https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md.

[50] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}? In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS'15)*.

[51] Batul J. Mirza, Benjamin J. Keller, and Naren Ramakrishnan. 2003. Studying recommendation algorithms by graph analysis. *J. Intell. Info. Syst.* 20, 2 (2003), 131–160.

[52] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 1–14.

[53] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 457–468.

[54] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 456–471. https://doi.org/10.1145/2517349.2522739

[55] The University of Texas at Austin. 2019. Texas Advanced Computing Center (TACC). Retrieved from https://www.tacc.utexas.edu/.

[56] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 166–177.

[57] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, New York, NY, 1–19. https://doi.org/10.1145/2983990.2984015

[58] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, New York, NY, 701–710. https://doi.org/10.1145/2623330.2623732

[59] The Lemur Project. 2013. The ClueWeb12 Dataset. Retrieved from http://lemurproject.org/clueweb12/.

[60] Meikang Qiu, Lei Zhang, Zhong Ming, Zhi Chen, Xiao Qin, and Laurence T. Yang. 2013. Security-aware optimization for ubiquitous computing systems with SEAT graph approach. *J. Comput. Syst. Sci.* 79, 5 (2013), 518–529.

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. Association for Computing Machinery, New York, NY, 472–488. https://doi.org/10.1145/2517349.2522740

[62] Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM'13)*. ACM, New York, NY, Article 22, 12 pages. https://doi.org/10.1145/2484838.2484843

[63] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 475–486. https://doi.org/10.1145/2485922.2485963

[64] Satu Elisa Schaeffer. 2007. Graph clustering. *Comput. Sci. Rev.* 1, 1 (2007), 27–64.

[65] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.* 6, 14 (Sep. 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

[66] Manjunath Shevgoor, Jung-Sik Kim, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Aniruddha N. Udipi. 2013. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 198–209.

[67] Julian Shun. 2019. K-Core. Retrieved from http://jshun.github.io/ligra/docs/tutorial_kcore.html.

[68] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 135–146. https://doi.org/10.1145/2442516.2442530

[69] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel local graph clustering. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1041–1052. https://doi.org/10.14778/2994509.2994522

[70] AM Stankovic and MS Calovic. 1989. Graph oriented algorithm for the steady-state security enhancement in distribution networks. *IEEE Trans. Power Delivery* 4, 1 (1989), 539–544.

[71] Lei Tang and Huan Liu. 2010. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*. Springer, 487–513.

[72] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Sotware-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 652–665.

[73] Keval Vora. 2019. LUMOS: Dependency-driven disk-based graph processing. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'19)*. USENIX Association, USA, 429–442.

[74] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery, New York, NY, 237–251. https://doi.org/10.1145/3037697.3037748

[75] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *Proceedings of the ACM International Conference on Object*

Oriented Programming Systems Languages & Applications (OOPSLA'14). ACM, New York, NY, 861–878. https://doi.org/10.1145/2660193.2660227

[76] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. 2011. Understanding graph sampling algorithms for social network analysis. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 123–128.

[77] English Wikipedia. 2013. enwiki-2013. Retrieved from http://law.di.unimi.it/webdata/enwiki-2013/.

[78] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15)*. ACM, New York, NY, 408–421. https://doi.org/10.1145/2806777.2806849

[79] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux2: Distributed graph computation for machine learning. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, 669–682.

[80] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. Association for Computing Machinery, New York, NY, 247–260. https://doi.org/10.1145/1629575.1629600

[81] Torsten Zesch and Iryna Gurevych. 2007. Analysis of the Wikipedia category graph for NLP applications. In *Proceedings of the TextGraphs-2 Workshop (NAACL-HLT'07)*. 1–8.

[82] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the hidden dimension in graph processing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 285–300. Retrieved from http://dl.acm.org/citation.cfm?id=3026877.3026900.

[83] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 544–557.

[84] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

[85] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 301–316. http://dl.acm.org/citation.cfm?id=3026877.3026901

[86] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. ACM, New York, NY, 712–725. https://doi.org/10.1145/3352460.3358256

Received July 2020; revised December 2020; accepted March 2021