SympleGraph: Distributed Graph Processing with Precise Loop-Carried Dependency Guarantee

Youwei Zhuo∗
University of Southern California
USA
youweizh@usc.edu

Jingji Chen∗
University of Southern California
USA
jingjich@usc.edu

Qinyi Luo
University of Southern California
USA
qinyiluo@usc.edu

Yanzhi Wang
Northeastern University
USA
yanz.wang@husky.neu.edu

Hailong Yang
Beihang University
China
hailong.yang@buaa.edu.cn

Depei Qian
Beihang University
China
depeiq@buaa.edu.cn

Xuehai Qian
University of Southern California
USA
xuehai.qian@usc.edu

Abstract
Graph analytics is an important way to understand relationships in real-world applications. At the age of big data, graphs have grown to billions of edges. This motivates distributed graph processing. 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. In essence, there exists a gap between programming model and runtime execution.

This paper proposes SympleGraph, a novel distributed graph processing framework that precisely enforces loop-carried dependency, i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. SympleGraph 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. Therefore, the major challenge is to enable sufficient parallelism to achieve high performance. We propose to use circulant scheduling in the framework to allow different machines to process disjoint sets of edges/vertices in parallel while satisfying the sequential requirement. It achieves a good trade-off between precise semantics and parallelism. The significant speedups in most graphs and algorithms indicate that the benefits of eliminating unnecessary computation and communication overshadow the reduced parallelism. Communication efficiency is further optimized by 1) selectively propagating dependency for large-degree vertices to increase net benefits; 2) double buffering to hide communication latency. In a 16-node cluster, SympleGraph outperforms the state-of-the-art system 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%.

CCS Concepts:
• Computing methodologies → Distributed programming languages.

Keywords: graph analytics, graph algorithms, compilers, big data

ACM Reference Format:

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 [57], natural language processing [2, 20, 59], anomaly detection [43, 50], clustering [46, 49], recommendation [16, 23, 37], social influence analysis [12, 51, 55], and bioinformatics [1, 14, 29].

At the age of big data, graphs have grown to billions of edges and will not fit into the memory of a single machine. Even if they can, the performance will be limited by the number of cores. Single-machine processing is not a truly scalable solution. To process large-scale graphs efficiently, a number of distributed graph processing frameworks have been proposed, e.g., Pregel [32], GraphLab [31], PowerGraph [18], D-Galois [13], and Gemini [61]. These frameworks partition the graph to distributed memory, so the neighbors of a vertex are assigned to different machines. To hide the details and complexity of distributed data partition and computation, these frameworks abstract computation 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.

The framework is responsible for scheduling computations and correctly performing communication and synchronization. To achieve good performance, both communication and computation need to be efficient. The communication problem, which is related to graph partition and replication, has been traditionally a key consideration of distributed framework, and prior works have proposed 1D [31, 32], 2D [18, 61], 3D [60] partition, and investigated the design space extensively [13]. This paper makes the first attempt to improve the efficiency of the two factors at the same time by reducing redundant computation and communication.

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 u1 and u2 of vertex v. If u1 satisfies an algorithm-specific condition, u2 will not be processed due to the dependency. The pattern appears in several important algorithms. Consider the bottom-up breadth-first search (BFS) [4] with pseudocode in Figure 1a. 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”, it will no longer traverse other neighbors and mark the vertex as “visited”.

In distributed frameworks [11, 13, 17, 18, 25, 27, 32, 45, 56, 61], programmers can write a break statement in UDF to indicate the control dependency. Figure 1b (b) shows signal-slot implementation of bottom-up BFS in Gemini [61]. The signal and slot UDF specify the computation to process each neighbor of a vertex and vertex update, respectively. We see 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 u1 and u2 are distributed in different machines, u1 and u2 can be processed in parallel and u2 does not know the state after processing u1. Therefore, the loop-carried dependency specified in UDF is not precisely enforced in the execution, thereby only an “illusion”.

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.

**Figure 1.** Bottom-up BFS Algorithm

```plaintext
1  def bfs(Array[Vertex] nbr) { 1  def signal(Vertex v, Array[Vertex] nbr)
2      for v in V { 2      for u in mbr {
3          for u in mbr { 3          if (not visited[v]) {
4              if (not visited[v]) { 4              if (frontier[u]) {
5                  frontier[v] = u; 5                  emit(v, u);
6                  parent[v] = u; 6                  break;
7                  visited[v] = true; 7                     }
8                  frontier[v] = true; 8                  break;
9 7                     }
10                     } 9                     }
11                     } 11                     }
12  // end for u 12 13  // end signal
13  } 13 14  def slot(Vertex v, Vertex upt) {
14 9 15  } 15 16  // end bfs
15  } 16 17 18  // end bottom_up_bfs
17  } 18
```

(a) Bottom-up BFS
(b) Bottom-up BFS in Gemini
To address this problem, we propose SympleGraph, a novel framework for distributed graph processing that enforces the loop-carried dependency in UDF. SympleGraph 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 machines process disjoint sets of edges and vertices. If one machine determines that the execution should break in a step, the break information is passed to the following machines so that the remaining neighbors are not processed. In practice, the computation and update communication of each step can be largely overlapped (see details in Section 5.3); thus the fine-grained steps do not introduce much extra overhead.

SympleGraph not only eliminates unnecessary computation but potentially reduces the total amount of communication. On the one side, small dependency messages, 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 enforcing loop-carried dependency can eliminate unnecessary computation and communication. Our results show that the total amount of communication is indeed reduced in most cases (Section 7.3, Table 6). To further reduce communication, SympleGraph differentiates dependency communication for high-degree and low-degree vertices, and only performs dependency propagation for high-degree vertices. We apply double buffering to enable computation and dependency communication overlapping and alleviate load imbalance.

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 [10]. We compare SympleGraph with two state-of-the-art distributed graph processing systems, Gemini [61] and D-Galois [13]. 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%.

2 Background and Problem Formalization

2.1 Graph and Graph Algorithm

Graph. 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 (BFS). BFS 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 [4] 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 (MIS). 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 [47] (Figure 3 (b)) 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. K-means is a popular clustering algorithm in data mining. Graph-based K-means [45] 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

The name SympleGraph does not imply symbolic execution. Instead, it refers to the key insight of scheduling the symbol execution order and making all evaluation concrete.

The source code of SympleGraph framework and execution scripts can be found in this link: https://github.com/zhuoyw/SympleGraph.

1There are other K-core algorithms with linear time complexity [44]. We choose this algorithm to demonstrate the basic code pattern. We also compare with the algorithm in evaluation.
We show an example of neighbor vertex sampling in Fig.

There are two design aspects of distributed graph framework:

(1) and repeat the algorithm.

master

Incoming edge-cut, a mirror vertex is generated if its
incoming edges are partitioned across all three machines,
so mirrors of v are created on machine 1 and 2.

outgoing neighbors of vertices, while pull mode traverses
outgoing neighbors. The five graph algorithms discussed
earlier are more efficient in pull mode in most iterations, and
symplegraph shows the pseudocode of pull mode. The signal function is
first executed on mirrors

and incoming edges of a vertex can be assigned to different
machines. It is used in PowerGraph [18] and Graphx [19].
Recent work [60] also proposed 3D graph partition that di-
vides 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 frame-
works. It is not applicable to incoming edge-cut, which will be
discussed in Section 2.3.

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
deges, 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 [48] 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


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.

(a) MIS
(b) K-core
(c) K-means
(d) Graph Sampling

Figure 3. Examples of algorithms with loop-carried dependency

and incoming edges of a vertex can be assigned to different
machines. It is used in PowerGraph [18] and GraphX [19].
Recent work [60] also proposed 3D graph partition that di-
vides 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 frame-
works. It is not applicable to incoming edge-cut, which will be
discussed in Section 2.3.

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
deges, 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 [48] 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

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, terminate the algorithm, else, goto
(1) and repeat the algorithm.

Graph Sampling. Graph sampling is an algorithm that
picks a subset of vertices or edges of the original graph.
We show an example of neighbor vertex sampling in Fig-
ure 3 (d), which is the core component of graph machine
learning algorithms, such as DeepWalk [41], node2vec [22],
and Graph Convolutional Networks [3]. In order 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. 4

2.2 Distributed Graph Processing Frameworks

There are two design aspects of distributed graph framework:
programing abstraction, and graph partition/replication.
Programming abstraction deals with how to express algo-
rithms with a vertex function. Graph partition determines
how vertives and edges are distributed, replicated, and syn-
chronized in different machines.

Master-mirror. To describe vertex replications, current frame-
works [11, 15, 18, 61] 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 [13]. Incoming edge-
cut: Incoming edges of one vertex are assigned only to
one machine, while its outgoing edges may be partitioned; Out-
going edge-cut: Outgoing edges of each vertex are assigned
only to one machine, while its incoming edges are parti-
tioned. It is used in several systems, including Pregel [32],
GraphLab [31], Gemini [61]. Vertex-cut: Both the outgoing
vertex after receiving all updates. Figure 2 also illustrated how the signal-slot function is applied for vertex 9. The blue edges (in machine 1 and 2) refer to signals, and the yellow edges (in machine 3) refer to slots. Green edges across machines indicate communication.

We can formalize the signal-slot abstraction by borrowing the notions of distributed functions in [58].

**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. \( C \) is commutative: \( \forall u_1, u_2, \ C(u_1 \oplus u_2) = C(u_2 \oplus u_1) \);
3. \( C \) is associative: \( \forall u_1, u_2, u_3, C(u_1 \oplus u_2) \oplus u_3) = C(u_1 \oplus C(u_2 \oplus u_3)) \).

Generally, all 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, a system 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 \( H \):

\[
\forall u_1, u_2, \ H(u_1 \oplus u_2) = C(I(u_1 \oplus u_2)) = C(I(u_1) \oplus I(u_2)).
\]

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

### 2.3 Inefficiencies with Existing Frameworks

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 paper) 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 \( \text{cost}(\cdot) \) to denote the computation cost of a function or the communication amount for the output of a function, a function \( I \) has redundancy without enforcing dependency if \( \forall u_1, u_2, \ \text{cost}(I(u_2)) \geq \text{cost}(I(u_2|u_1)) \) and \( \exists u_1, u_2, \ \text{cost}(I(u_2)) > \text{cost}(I(u_2|u_1)) \).

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, 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 [11, 13, 17, 18, 25, 27, 32, 45, 56, 61] precisely enforce loop-carried dependency semantics. While the incoming edge-cut is an exception, the partition is inefficient and rarely used due to load imbalance issues. According to D-Galois (Glou), they used the vertex-cut partition by default “since it performs well at scale” [13].
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, 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, 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 SympleGraph Overview

We propose SympleGraph, a new distributed graph processing framework that precisely enforces loop-carried dependency semantics in UDFs. SympleGraph workflow consists of two components. The first one is UDF 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 machines.

The second component is system support 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 machines 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 (Section 5.1).

4 SympleGraph Analysis

4.1 SympleGraph Primitives

SympleGraph provides dependency communication primitives, which are used internally inside the framework and transparent to programmers. 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 5 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 (Line 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 SympleGraph 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.

\footnote{The source code of SympleGraph analyzer can be found in this link: \url{https://github.com/AmadeusChan/SympleGraphAnalyzer}.}
4.3 Discussion

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

**New Graph DSL.** Besides the analysis, SympleGraph provides a new 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.

5 SympleGraph System

In this section, we discuss how SympleGraph schedules dependency communication to enforce 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 6, 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, 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 \(\sigma\) is defined as \(\sigma(i) = (i + p - 1) \mod p\), and initially \(\sigma(i) = i, i = 0, ..., (p - 1)\). The vertices in a graph is divided into \(p\) disjoint sets according to \(\sigma\) master vertices. Let \(u^{(i)}\) denote the sequence of neighbors of master vertices on machine \(i\). In step \(j (j = 0, 1, ..., p - 1)\), circulant scheduling executes \(I(u^{(j)})\) on machine \(\sigma^{(j)}\).

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 \(I(u_{\sigma^{(1)}} \rightarrow u_{\sigma^{(2)}} \rightarrow ... \rightarrow u_{\sigma^{(p-1)}})\), starting from step 0. For any specific step \(j\), the scheduling specifies different machines,
because $\sigma^j$ 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 7 shows an example with four machines. Figure 7 (a) shows the matrix view of the graph. An element $(i,j)$ in the matrix represents an edge $(v_i,v_j)$. Similarly, we use the notion $[i,j]$ to represent a grid, i.e., the 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 grid $[i,j]$ in parallel. For example, in step 0, the grids processed by machine $0,1,2,3$ are $[0,1],[1,2],[2,3],[3,0]$, respectively. Across all steps, edges belonging to all grids $[j,i], j \in \{0,1,2,3\}$, are processed sequentially.

Figure 7 (b) shows the step execution according to Figure 7 (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 execution. Without circulant scheduling, a machine has the freedom to process all edges with sources allocated to this machine (a range of rows in Fig6 (a)); with circulant scheduling, during a given step (a part of an iteration), the machine can only process edges in the corresponding grid. 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 7 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 8 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 machine 2 and machine 4, even when there is no mirror of the vertex on machine 3, we still need to send a message from machine 2 to 3 because we cannot discard any message in a circulant ring 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, SympleGraph falls 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 to the original scheduling 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 ends up not skipping many neighbors during execution. PowerLyra [11] proposed differentiated graph partition optimization that reduces update communication for low-degree vertices. In SympleGraph, differentiation is relevant to update communication, and it is orthogonal to graph partition.

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 are overlapped as shown in Figure 7, the dependency communication still 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 9 demonstrates the key idea with an example. We consider two machines and the first two steps. Specifically, the figure shows the dependency 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 directly 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 due to reduced load imbalance. Consider the potential scenario of load imbalance in Figure 9, machine 3 (receiver) has much less load in step 1 and proceeds to the next step before machine 1. Without double buffering, machine 3 has to wait for the full completion of machine 1 in step 1. In the double buffering, it only waits for the dependency message of group A. Since that message is sent first, it is likely to have already arrived.

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.

6 Implementation Details

SympleGraph is implemented using C++ based on Gemini. The key system component is the dependency communication library, a dynamic library that works for all distributed graph processing frameworks.

To support dependency primitives, SympleGraph builds dependency data structure from data fields in struct DepMessage. 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, accessed by emit_dep and receive_dep.

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 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, the latency of dependency will increase. If we send the message too frequently, the worker-coordinator synchronization overhead becomes intolerable. In SympleGraph, we choose to batch the message by 16 KB.

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 pre-processing 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, 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 and interference on the receiver side.

7 Evaluation

We evaluate SympleGraph and two state-of-the-art distributed graph processing frameworks, Gemini [61], and D-Galois [13]. We choose Gemini because we implement the proposed techniques based on its signal-slot interface, thus the comparison can directly demonstrate the performance benefits. D-Galois is a recent framework 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.

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 (56Gb/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 [39]. Each node has 2 Intel Xeon Platinum 8160 (24 cores/CPU) and 192 GB DRAM with 100Gb/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 256GB memory. The network is InfiniBand FDR (56Gb/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 [10]. We use the same generator parameters as in Graph500 benchmark [21].

Table 1. Graph datasets. $|V'|$ is the number of high-degree vertices

| Graph          | Abbrev. | $|V|$ | $|E|$ | $|V'|/|V|$ |
|----------------|---------|------|------|---------|
| Twitter-2010   | tw      | 42M  | 1.5B | 0.13    |
| Friendster     | fr      | 66M  | 1.8B | 0.31    |
| R-MAT-Scale27-E32 | s27       | 134M | 4.3B | 0.12    |
| R-MAT-Scale28-E16 | s28       | 268M | 4.3B | 0.09    |
| R-MAT-Scale29-E8 | s29       | 537M | 4.3B | 0.04    |
| Clueweb-12 [8, 42] | cl       | 978M | 43B  | 0.12    |
| Gsh-2015 [7]   | gsh     | 988M | 34B  | 0.28    |
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 [48] that chooses from both top-down and bottom-up algorithms in each iteration. 6 7 We follow the optimization instructions in D-Galois by running all partition strategies provided and report the best one as the baseline. 8

For BFS, we average the experiment results of 64 randomly generated non-isolated roots. For each root, we run the algorithm 5 times. For K-core, 2-core is a subroutine widely used in strongly connected component [26] algorithm. We also evaluate other values of K. For K-means, 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× 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 [34]. 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. We see that the speedups are not affected by K.

---

8 Adaptive switch is not available in D-Galois. For fair comparison, we implement the same switch in D-Galois.

7 Graph sampling implementation is not available in D-Galois.

6 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.

9 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.

<table>
<thead>
<tr>
<th>Graph</th>
<th>App.</th>
<th>Gemini</th>
<th>SympleG.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>tw</td>
<td>K-core</td>
<td>24.1753</td>
<td>13.4465</td>
<td>1.80</td>
</tr>
<tr>
<td>fr</td>
<td>K-means</td>
<td>84.7207</td>
<td>75.7227</td>
<td>1.12</td>
</tr>
<tr>
<td>fr</td>
<td>Sampling</td>
<td>4.6578</td>
<td>3.4686</td>
<td>1.34</td>
</tr>
</tbody>
</table>

### 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...
Table 4. Execution Time (in seconds)

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

Table 5. Number of traversed edges (Normalized to total number of edges in the graph)

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

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 10). 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...
Table 6. SympleGraph communication breakdown (Normalized to total communication volume in Gemini)

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

COST. The COST metric [36] 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 Ga- lois [38] and s27 graph as the single-thread baseline. The COST of Gemini and SympleGraph is 4, while the COST of D-Galois is 64. We also use the BFS algorithm in GAPBS [5] and tw graph as another baseline. GAPBS finishes in 2.29 seconds while SympleGraph takes 2.66 and 1.83 seconds 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. Thus, we believe using SympleGraph on a small-scale distributed cluster is the most convenient and practical solution.

Table 7. Execution time (in seconds) of MIS using the best-performing number of nodes (in parenthesis) on Stampede2

<table>
<thead>
<tr>
<th>Graph</th>
<th>D-Galois</th>
<th>SympleGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>tw</td>
<td>1.321(128)</td>
<td>1.113(2)</td>
</tr>
<tr>
<td>fr</td>
<td>1.355(128)</td>
<td>0.823(4)</td>
</tr>
<tr>
<td>s27</td>
<td>1.258(128)</td>
<td>0.911(4)</td>
</tr>
<tr>
<td>s28</td>
<td>1.380(128)</td>
<td>1.159(4)</td>
</tr>
<tr>
<td>s29</td>
<td>1.565(128)</td>
<td>1.420(4)</td>
</tr>
</tbody>
</table>

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

8 Related Work

BFS Systems [6, 9] 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) [44] 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** [33, 52–54] proposes 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 [40] and Abelian [17] 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 (DSL).** Some DSLs (e.g., GreenMarl [24] and GRAPE [15]) capture algorithm information by asking the users to program in a new programming interface that can express new semantics. For example, GRAPE describes graph algorithms with “partial evaluation”, “incremental evaluation” and “combine”. GRAPE system implementation is not efficient: the reported distributed performance on 24 machines is worse than single-thread naive implementation on a laptop [35].

![Figure 11. Analysis of optimizations (baseline is SympleGraph with only circulant scheduling)](image)

**9 Conclusion**

This paper proposes SympleGraph, a novel framework for distributed graph processing that precisely enforces loop-carried dependency, i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. SympleGraph analyzes user-defined functions and identifies the loop-carried dependency. The distributed framework enforces the precise semantics by performing dependency propagation dynamically. To achieve high performance, we apply circulant scheduling in the framework to allow different machines to process disjoint sets of edges and vertices in parallel while satisfying the sequential requirement. To further improve communication efficiency, SympleGraph differentiates dependency communication and applies double buffering.

In a 16-node setting, 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%.

**Acknowledgments**

We would like to thank the anonymous reviewers for their helpful feedback. This work is supported by National Science Foundation (Grant No. CCF-1657333, CCF-1717754, CNS-1717984, CCF-1750656, CCF-1919289). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562 through allocation CCR190022. We used the Stampede2 system at the Texas Advanced Computing Center (TACC).

**References**
