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

Youwei Zhuo  
University of Southern California  
youweizh@usc.edu

Jingji Chen  
University of Southern California  
jingjich@usc.edu

Qinyi Luo  
University of Southern California  
qinyiluo@usc.edu

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

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

Depei Qian  
Beihang University  
depeiq@buaa.edu.cn

Xuehai Qian  
University of Southern California  
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. Even when the graph itself can fit in storage or memory of a single machine, its compute resource is limited. 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, it indicates a gap between programming model and runtime execution.

This paper proposes SympleGraph, a novel workflow 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 the UDFs of unmodified codes, identifies, and instruments the codes to express the loop-carried dependency. The distributed framework enforces the precise semantics by performing dependency propagation dynamically. Enforcing loop-carried dependency requires that all neighbors of a vertex are processed sequentially, thus the major challenge is how to still enable sufficient parallelism to achieve high performance. We apply circulant scheduling in the framework to allow different machines to process disjoint sets of edges/vertices in parallel while satisfying the sequential requirement. 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; and 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 on average is 40.95% and up to 67.48%.

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, 19, 58], anomaly detection [43, 49, 55], clustering [45, 48], recommendation [15, 22, 30, 37], social influence analysis [9, 50, 54], and bioinformatics [1, 13, 28].

At the age of big data, graphs have grown to billions of edges and will not likely fit into the memory of a single machine. Even if they can, the performance will be limited by the number of cores in one machine. It is not a truly scalable solution. To process large-scale graphs efficiently, a number of distributed graph processing frameworks have been proposed, for example, Pregel [32], GraphLab [31], PowerGraph [17], D-Galois [12], and Gemini [60]. To hide the details and complexity of distributed computation, these frameworks abstract computation as vertex-centric User-Defined Functions (UDFs) P(v), which is executed in parallel on each vertex v. In each P(v), programmers can access the neighbors of v as if they are local.

With graph partition, since the neighbors and edges of a vertex can be assigned to different remote machines, UDFs (computation) can be executed in parallel with multiple machines. The framework is responsible for scheduling computations and efficiently handling communication and synchronization. To achieve good performance, both communication and computation need to be done efficiently. The communication problem, which is related to graph partition and replication, has been traditionally been a key consideration of distributed framework, and prior works have proposed 1D [31, 32], 2D [17, 60], 3D [59] partition, and investigated the design space [12]. To achieve efficient computation, recent works [33, 51–53] on asynchronous graph processing uses relaxed dependency of values between iterations. However, as we will discuss shortly, SympleGraph is the first framework that enforces dependency within one iteration.
A common code pattern used in UDFs is *loop-carried dependency*: the UDF is *stateful* when traversing the neighbors of a vertex in a loop. The semantics is that the processing of each neighbor follows the specified dependency order. Due to the nature of distributed execution, the execution behavior can be different. Specifically, consider two neighbors $n_1$ and $n_2$ of vertex $v$, according to the dependency, if $n_1$ satisfies a certain condition, $n_2$ will not be processed. If the processing of $n_1$ and $n_2$ are distributed in different machines, they can process $n_1$ and $n_2$ in parallel and $n_2$ does not know the state after processing $n_1$. Therefore, the loop-carried dependency specified in UDF is not faithfully enforced.

This pattern exists in several important algorithms. Let us consider the bottom-up breadth-first search (BFS) [4] 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”, it will no longer traverse other neighbors and mark the vertex as “visited”. Compared to top-down BFS, bottom-up BFS avoids the inefficiency due to multiple visits of one new vertex in the frontier. According to [4], it significantly reduces the number of edges traversed.

Figure 1 (b) shows signal-slot implementation of bottom-up BFS in Gemini [60]. The *signal* and *slot* UDF specify the computation to process each neighbor of a vertex and vertex property 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. In distributed frameworks [11, 12, 16, 17, 24, 26, 32, 44, 45, 60], although programmers can write such break statement in UDF, as explained earlier, it is only an “illusion”.

The consequence of such imprecise execution behavior is the *unnecessary computation and communication* between machines. 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 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. The *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 *SympleGraph* [12], a novel workflow for distributed graph processing that enforces precise loop-carried dependency semantics. Specifically, SympleGraph analyzes the UDFs of unmodified codes, identifies, and instruments the codes to express the loop-carried dependency. The distributed framework enforces the precise semantics by performing dependency propagation dynamically. 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 guaranteeing the sequential requirement, we apply *circulant scheduling* and divide the execution of each iteration is divided 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 5.4), thus the fine-grained steps do not introduce much extra overhead.

SympleGraph not only eliminates unnecessary computation, but may even reduce the total amount of communication—composed of update and dependency messages. On 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 (large) update communication. Our results show that total

---

1 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.

2 The source code of SympleGraph framework and execution scripts can be found in this link: https://anonymous.4open.science/repository/ch220d2c-2546-41f4-8d89-043aa1a5c856/
amount of communication is indeed reduced in most cases (Section 7.3, Table 6). To further reduce total 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 [60] and D-Galois [12]. 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

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 API 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".

The 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 top-down approach, bottom-up BFS can significantly reduce 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 [46] (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 [44] 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, 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 Figure 3 (d), which is the core component of graph machine learning algorithms, such as DeepWalk [41], node2vec [21], 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.

2.2 Distributed Graph Processing Frameworks

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

Master-mirror. To describe vertex replications, current frameworks [11, 12, 17, 60] 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. Incoming edge-cut: Incoming edges of one vertex are assigned only to one machine (the master node), while its
Outgoing edges are partitioned. Outgoing edge-cut: Outgoing edges of each vertex are assigned only to one machine (the master node), while its incoming edges of a node are partitioned. It is used in several systems, including Pregel [32], GraphLab [31], Gemini [60]. Vertex-cut: Both the outgoing and incoming edges of a vertex are assigned to different machines. It is used in PowerGraph [17] and GraphX [18]. Recent work [59] also proposed 3D graph partition that divides the vector data of vertices into layers. This dimension is orthogonal to other partition methods for two dimensions.

In outgoing edge-cut, a mirror vertex will be generated if one of its incoming neighbors is a master vertex on the machine. 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. In this example, vertex 9 has 8 incoming edges. The sources are vertex 1 to 8. Vertex 1 to 3 are masters on machine 1, and vertex 4 to 6 are masters on machine 2. Vertex 9 is master on machine 3, so mirrors of v are created on machine 1 and 2.

**Signal-Slot.** As discussed in Ligra [47], there are two modes of signal-slot: push and pull. Push mode traverses and updates the out-going neighbors of vertices. In contrast, pull mode traverses the in-coming neighbors. The five graph algorithms discussed earlier are more efficient in pull mode in most iterations. SympleGraph optimization will focus on pull mode.

**Figure 3.** Examples of algorithms with loop-carried dependency

![Graph Sampling](image)

2.3 Inefficiencies with Existing Frameworks

Current vertex-centric programming abstractions like signal-slot are flexible enough for users to express many operations on vertex neighbors. However, loop-carried dependency is invisible to the framework. Even if programmers can specify dependency in UDF, the UDF runs on mirrors on different machines in parallel. The state of one UDF is not available to other machines running the same UDF.

When loop-carried dependency is not faithfully enforced by the framework, it will lead to unnecessary computation and communication. In Figure 2, we can count the number of edges traversed and communication size. The circles with colors are incoming vertices that satisfy the break conditions in bottom-up BFS. In machine 1, the signal function stops traversing after vertex 1, so vertex 2 and vertex 3 are skipped. In machine 2, it iterates all 3 vertices. However, according to the semantics, all vertices in machine 2 should not have been processed, it fails to recognize that opportunity because machine 2 is not aware of the dependency in machine 1.

**Applicability** Unless with incoming edge-cut, i.e., all of the incoming edges of one vertex are on the same machine, the execution of UDFs is always distributed. Therefore, the problem is not specific to out-going edge cut. To our knowledge, none of distributed systems [11, 12, 16, 17, 24, 26, 32, 44, 56, 60] precisely enforce loop-carried dependency semantics. While incoming edge-cube does not suffer from the problem, few distributed systems implement this partition due to its load imbalance issue. According to D-Galois (Gluon), they used a vertex-cut partition by default “since it performs well at scale”.

**Figure 4.** Signal-Slot in Pull Mode

![Signal-Slot in Pull Mode](image)
the neighbors is assigned to the nearest cluster center, the vertex can be assigned the same center. **Graph sampling** has data and control dependency. The sample is dependent on the random number and all the preceding neighbor 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 the building blocks of other more complicated algorithms.

### 3 SympleGraph Overview

We propose SympleGraph, a new distributed graph processing workflow 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 UDF analyzed 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 edges 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 5.2).

### 4 SympleGraph Analysis

#### 4.1 SympleGraph Primitives

SympleGraph provides dependency communication primitives, which are intended to be used internally and transparent to programmers. Dependent message has a data type DepMessage with two types of data members: a bit for 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\_t\_dep\_t and receive\_t\_dep\_t send and receive the dependency state of a vertex, where the type of T is DepMessage. We first describe how does SympleGraph use 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 using bottom-up BFS with dependency information and primitive. When processing a vertex u, the framework first executes emit\_t\_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\_t\_dep is inserted to notify the next machine which executes the function. Note that emit\_t\_dep and emit\_t\_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 simplify the analyzer design, we make domain-specific assumptions on graph processing UDFs. SympleGraph analyzer\(^3\) is based on Gemini’s signal-slot programming abstraction, but our assumptions are all inherent in graph algorithms, thus the ideas of SympleGraph analyzer are general to other graph processing frameworks. Specifically, we make the following assumptions:

- The graph algorithm logic is written in the UDFs.
- All UDFs are defined using lambda expressions that capture variables. Copy statements of these variables are not allowed so that we can locate the UDFs and variables.
- The UDF traverses 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\_t\_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 5.

#### 4.3 Discussion

In this section, we discuss the alternative approaches that can be used to enforce loop-carried dependency and the relevance of our solution to the PL community.

---

\(^3\) The source code of SympleGraph analyzer can be found in this link: [https://anonymous.4open.science/repository/d4111701-582d-44fc-a72f-0200cf725059/](https://anonymous.4open.science/repository/d4111701-582d-44fc-a72f-0200cf725059/).
visited array as the break dependency state. The visited we record the dependency for a vertex, the visited has al-

denies a new DSL and asks the programmer to describes loop

Figure 5. SympleGraph instrumented bottom-up BFS UDFs

New Graph DSL. Besides the analysis, SympleGraph pro-

vides a new DSL and asks the programmer to describes 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 func-
tion that composes dependency state and current neighbor,
a condition that exits the loop. The compiler can easily deter-
dine 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 pro-
grammers can manually analyze and optimize the code. Sym-
pleGraph also exposes communication primitives to the pro-
grammers so that they can still leverage the optimizations
when the code is not amenable 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 communica-
tion 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 al-
ready been set, so we can reduce computation by avoiding
the bit set operation in dependency bitmap. When we send
the dependency, we can actually send "visited" and avoid
the memory allocation for dependency communication.

However, we believe 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 almost every graph algorithm, the data
per vertex takes at least four bytes. As for the computation
reduction, setting a bit sequentially in a bitmap is also a
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
weaken the readability of the source code, increase the bur-
den of the user, and hurt programmability. 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 repur-
pose "frontier" as the dependency data.

Relevance to PL Community. With the assumptions,
SympleGraph analyzer appears to be not sophisticated. De-
tecting these dependencies may be difficult for general C++
codes. However, we do not claim the analyzer as our main
contribution. Instead, the key point we demonstrate in the
paper is by simple program analysis and co-design the in-
strumented codes with the framework, significant perform-
ance improvement for distributed graph processing can
be achieved. Future works can be conducted to find more
patterns from more general codes. Still, for the first time, this
paper shows the mismatch of program semantics and dis-
tributed execution of graph algorithms. We believe that the
new PL perspective will encourage similar optimizations in
the future.

While it is not a typical PL paper, the recent work Gluon [12]
on the same topic (communication in distributed graph pro-
cessing) also appeared in PLDI’18. Gluon also compares the
performance with Gemini, and shows its best performance
(that happens with 128 nodes) is better than Gemini’s best
performance with 16 nodes. This paper has achieved much
more significant results: 1) SympleGraph directly improves
Gemini’s performance using the same number of nodes; 2)
SympleGraph on a local cluster with 4 nodes can achieve the
performance of D-Galois using 128 nodes.

5 SympleGraph System

In this section, we discuss the implementation of communi-
cation primitives, enforcing the dependency with efficient
system support.

5.1 SympleGraph Primitives Implementation

To support dependency primitives, SympleGraph builds de-
pendency 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_depth and
receive_depth.

5.2 Enforcing Dependency: Circulant Scheduling

To clarify our ideas, Figure 6 (a) shows the matrix view of
a graph distributed in four machines. An element (i,j) in
the matrix represents an edge (vi, vj). Similarly, we use the
notion [i,j] to represent a grid—the set of edges from machine
i to machine j. The sequential requirement means that all

```c
struct DepBFS : DepMessage { // datatype
    bit skip;
};

def signal (Vertex v, Array[Vertex] nbrs) {
    DepBFS d = receive_dep(v); // new code
    if (d.skip) {
        return;
    }
    foreach u in nbrs(v) {
        if (not visited(v)) {
            parent[v] = upt;
        }
        visited[v] = true;
    }
}

def slot(Vertex v, Vertex upt) {
    if (not visited(v)) {
        parent[v] = upt;
        visited[v] = true;
        frontier[v] = true;
    }
}
```
edges in the same column need to be processed sequentially across all machines. For iterative algorithms, one iteration processes all edges in the graph. Let us divide an iteration into multiple steps. In each step, machine $i$ processes all edges in $[i, j], j \in \{0, 1, 2, 3\}$. Figure 6 (a) shows one possible scheduling of steps that enables sufficient parallelism while still preserving the sequential execution requirement.

We call the execution in Figure 6 (a) circulant scheduling. The work for each machine is indicated in the range of rows corresponding to this machine. The machine will process grids according to the step order. For example, machine 0 first processes all edges in $[0, 1]$ and then $[0, 2], [0, 3], [0, 0]$. The other machines are similar. The key property of circulant scheduling is that the same step presents diagonally in the matrix view. As a result, during 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. The essential insight is that circulant scheduling can enable sufficient parallelism while still preserving the sequential execution requirement.

With computations distributed in machines by circulant scheduling, the framework propagates the dependency with communication primitives from one machine to another by dependency communication, a new type of communication SympleGraph introduces to enforce precise loop-carried dependency. Figure 6 (b) shows a clear view of synchronized step execution according to Figure 6 (a) with dependency communication. The communication pattern is the same for all steps. Even with circulant scheduling, the execution is still more restrictive than arbitrary execution, but it elegantly enables more parallelism by letting each machine process disjoint sets of edges in parallel. The significant improvements in Section 7 indicate that the eliminated redundant communication and computation can fully offset the effects of reduced parallelism with circulant scheduling.

Figure 6 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 precisely enforce dependency. For update communication, during the whole iteration, 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 miss some opportunities to eliminate unnecessary computation and communication. In fact, Gemini can be considered as a special case without dependency communication.

5.3 Differentiated Dependency Propagation

This section discusses an optimization to reduce total 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 7 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 \( v \) 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 less useful dependency communication, we propose to differentiate the dependency communication for high-degree and low-degree vertices. The degree threshold is a constant determined empirically. 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 less neighbors, the redundant computation due to loop-carried dependency is also insignificant, because it ends up not skipping many neighbors during execution. PowerLyra \cite{PowerLyra} 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.

### 5.4 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 6, 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 in machine 1 and 2. However, it still impairs performance. Besides the extra latency due to waiting for the dependency message, it also incurs load imbalance within the step. While the load balance problem has been studied extensively, the objective of all existing load balance techniques is to enhance load balance of the entire iteration. With steps and dependency communication, load imbalance may occur in each step. 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 8 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 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 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 8, machine 3 (receiver) has much less load in step 1 and proceeds to the next step before machine 1. Without the optimization, 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 the two
groups. Since processing low-degree vertices will 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.

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 supports NUMA-aware optimizations. If there is more than one NUMA node on the machine, we spawn more coordinators and initialize the data structure based on NUMA topology.

To leverage multi-core hardware in each machine, distributed graph processing frameworks, starts several 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 power of two numbers with 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 [60] and D-Galois [12]. 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 and the MPI library is OpenMPI 3.0.1. The network is Mellanox InfinitiBand FDR (56Gb/s). The following evaluation results are conducted in Cluster-A unless otherwise stated. (2) Cluster-B is Stampedede2 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 InfinitiBand 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 [20].

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 algorithms requiring un-directed graph using directed dataset, we consider every directed edge as its un-directed counterpart. To run algorithm on directed graphs, we convert the un-directed 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. 6 7 We follow the optimization instructions in D-Galois by running all partition strategies provided and report the best one as baseline. 8

---

6D-Galois does not support adaptive switch optimization between top-down and bottom-up mode [47]. We implement the optimization as the baseline.
7D-Galois graph sampling implementation is not available.
8We 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).
Table 1. Graph datasets. $|V'|$ is the number of high-degree vertices.

| Graph               | Abbrev. | $|V'|$ | $|E|$ | $\frac{|V'|}{|E|}$ |
|---------------------|---------|------|------|-----------------|
| Twitter-2010 [27]   | tw      | 42M  | 1.5B | 0.13            |
| Friendster [29]     | 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 [7, 42]  | cl      | 978M | 43B  | 0.12            |
| Gsh-2015 [6]        | gsh     | 988M | 34B  | 0.28            |

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

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 [25] and 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 other test cases, 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 a speedup up to 3.05x (1.46× on average) 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 has no loop dependency [34]. It is slower than SympleGraph for large synthesised 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 graphs have relatively small diameters, they usually have a long link structure attached to the small-diameter core parts.

**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.

**Large Graphs.** We run Gemini and SympleGraph with the two large real-world graphs (Clueweb-12 and Gsh-2015) on Cluster-C. SympleGraph has no improvement for BFS and K-means in Clueweb-12, because most of the iterations do not use bottom-up algorithm optimized by SympleGraph. For 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 forcing loop-carried dependency. In graph processing, the number of edges traversed is the most significant part in 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 69.91% 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 (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 21.7% edges compared with Gemini, while on $s29$ traverses 29.0%. When the graph structure is similar (R-MAT), the number of traversed edges is less in graphs with larger average degree. Large
Table 4. Execution Time (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Graph</th>
<th>Gemini</th>
<th>D-Galois</th>
<th>SympleG.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>0.608</td>
<td>2.053</td>
<td><strong>0.264</strong></td>
<td>2.30</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>1.212</td>
<td>4.993</td>
<td><strong>0.706</strong></td>
<td>1.72</td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>1.054</td>
<td>2.681</td>
<td><strong>0.733</strong></td>
<td>1.44</td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>1.325</td>
<td>3.682</td>
<td><strong>0.976</strong></td>
<td>1.36</td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>1.760</td>
<td>5.356</td>
<td><strong>1.372</strong></td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>K-core</td>
<td></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><strong>2.190</strong></td>
<td>1.38</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>11.258(0.580)</td>
<td>17.213</td>
<td><strong>7.390</strong></td>
<td>1.52</td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>2.754</td>
<td>3.512</td>
<td><strong>1.640</strong></td>
<td>1.38</td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>4.432</td>
<td>6.056</td>
<td><strong>2.663</strong></td>
<td>1.66</td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>5.413</td>
<td>8.534</td>
<td><strong>3.806</strong></td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>MIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>2.081</td>
<td>4.056</td>
<td><strong>1.421</strong></td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>2.363</td>
<td>5.045</td>
<td><strong>1.754</strong></td>
<td>1.35</td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>2.720</td>
<td>5.329</td>
<td><strong>1.861</strong></td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>3.031</td>
<td>7.110</td>
<td><strong>2.408</strong></td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>3.600</td>
<td>8.620</td>
<td><strong>2.835</strong></td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>K-means</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>17.590</td>
<td>56.748</td>
<td><strong>12.688</strong></td>
<td>1.39</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>19.212</td>
<td>78.526</td>
<td><strong>13.143</strong></td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>27.626</td>
<td>61.598</td>
<td><strong>19.279</strong></td>
<td>1.43</td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>34.393</td>
<td>86.632</td>
<td><strong>26.919</strong></td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>52.087</td>
<td>116.307</td>
<td><strong>41.760</strong></td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>Sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td><strong>0.786</strong></td>
<td>0.867</td>
<td>0.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>1.180</td>
<td>0.977</td>
<td>1.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>1.388</td>
<td>N/A</td>
<td><strong>1.090</strong></td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>2.051</td>
<td>1.331</td>
<td>1.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>2.932</td>
<td>1.869</td>
<td>1.57</td>
<td></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></th>
<th>Graph</th>
<th>Gemini</th>
<th>SympleG.</th>
<th>D-Galois</th>
<th>SympleG./Gemini</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>0.438</td>
<td>0.1562</td>
<td>0.3563</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>0.853</td>
<td>0.3116</td>
<td>0.3650</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>0.308</td>
<td>0.0402</td>
<td>0.1300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>0.358</td>
<td>0.0555</td>
<td>0.1549</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>0.471</td>
<td>0.0978</td>
<td>0.2074</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>3.642</td>
<td>1.1986</td>
<td>0.4537</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>11.328</td>
<td>3.1951</td>
<td>0.2820</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>1.118</td>
<td>0.3498</td>
<td>0.3126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>1.871</td>
<td>0.6165</td>
<td>0.3294</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>2.423</td>
<td>1.0513</td>
<td>0.4338</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>5.443</td>
<td>2.0479</td>
<td>0.3762</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>3.132</td>
<td>0.8717</td>
<td>0.2782</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>3.439</td>
<td>1.0174</td>
<td>0.2958</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>3.776</td>
<td>1.1970</td>
<td>0.3170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>3.901</td>
<td>1.9750</td>
<td>0.5062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-means</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>13.397</td>
<td>5.5608</td>
<td>0.4151</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>2.579</td>
<td>1.8989</td>
<td>0.7361</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>5.616</td>
<td>1.7196</td>
<td>0.3062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>8.835</td>
<td>2.7847</td>
<td>0.3152</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>13.647</td>
<td>5.3375</td>
<td>0.3911</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>1.031</td>
<td>0.2143</td>
<td>0.2078</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>1.209</td>
<td>0.1290</td>
<td>0.1066</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>1.109</td>
<td>0.0709</td>
<td>0.0639</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s28</td>
<td>1.149</td>
<td>0.0966</td>
<td>0.0840</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s29</td>
<td>1.191</td>
<td>0.1172</td>
<td>0.0984</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

average degree means more high-degree vertices that SympleGraph optimize in differentiated computation. Therefore, s27 have 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 9). 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 achieve the best performance with 8 machines. D-Galois scales to 16 machines, but its best performance requires 128 to 256 machines according to [12]. In summary, SympleGraph is consistently better than Gemini and D-Galois with 16 machines. 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 [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 Galoisa [38] and graph s27 as the single-thread baseline. The COST of Gemini and SympleGraph is 4, while the COST of D-Galois is 64.

D-Galois. To evaluate the best performance of D-Galois, We reproduce the results with Cluster-B. The results are shown.
Table 6. SympleGraph communication breakdown (Normalized to total communication volume in Gemini)

<table>
<thead>
<tr>
<th></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>0.0193</td>
<td>0.4326</td>
</tr>
<tr>
<td>s28</td>
<td>0.4855</td>
<td>0.0175</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>0.0039</td>
<td>0.5067</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>

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

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 10 only shows the average result over all algorithms. For each graph dataset, we normalize the runtime to the version with basic circulant scheduling.

Double buffering is effective 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 [5, 8] 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.

Graph compiler. IrGL [40] and Abelian [16] 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 considered in graph compilers.

Graph Domain Specific Language (DSL). Some DSLs (e.g., GreenMarl [23] and GRAPE [14]) 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", etc.

### 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>
“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].

9 Conclusion
This paper proposes SympleGraph, a novel workflow 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 the UDFs of unmodified codes, identifies, and instruments the codes to express 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/vertices in parallel while satisfying the sequential requirement. To further improve communication efficiency, SympleGraph differentiates the 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 on average are 40.95%, and up to 67.48%.

Acknowledgments
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


SympleGraph: Distributed Graph Processing with ...