# Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics

Roshan Dathathri*
Department of Computer Science
University of Texas at Austin, USA
roshan@cs.utexas.edu

Gurbinder Gill*
Department of Computer Science
University of Texas at Austin, USA
gill@cs.utexas.edu

Loc Hoang
Department of Computer Science
University of Texas at Austin, USA
l_hoang@utexas.edu

Hoang-Vu Dang
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
hdang8@illinois.edu

Alex Brooks
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
brooks8@illinois.edu

Nikoli Dryden
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
dryden2@illinois.edu

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
snir@illinois.edu

Keshav Pingali
Department of Computer Science
University of Texas at Austin, USA
pingali@cs.utexas.edu

## Abstract

This paper introduces a new approach to building distributed-memory graph analytics systems that exploits heterogeneity in processor types (CPU and GPU), partitioning policies, and programming models. The key to this approach is Gluon, a communication-optimizing substrate.

Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run on heterogeneous clusters and optimizes communication in a novel way by exploiting structural and temporal invariants of graph partitioning policies.

To demonstrate Gluon's ability to support different programming models, we interfaced Gluon with the Galois and Ligra shared-memory graph analytics systems to produce distributed-memory versions of these systems named D-Galois and D-Ligra, respectively. To demonstrate Gluon's ability to support heterogeneous processors, we interfaced Gluon with IrGL, a state-of-the-art single-GPU system for graph analytics, to produce D-IrGL, the first multi-GPU distributed-memory graph analytics system.

Our experiments were done on CPU clusters with up to 256 hosts and roughly 70,000 threads and on multi-GPU clusters with up to 64 GPUs. The communication optimizations in Gluon improve end-to-end application execution time by ~2.6× on the average. D-Galois and D-IrGL scale well and are faster than Gemini, the state-of-the-art distributed CPU graph analytics system, by factors of ~3.9× and ~4.9×, respectively, on the average.

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

***Keywords*** Distributed-memory graph analytics, communication optimizations, heterogeneous architectures, GPUs, big data

**ACM Reference Format:**
Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3192366.3192404

*Both authors contributed equally.

*PLDI'18, June 18–22, 2018, Philadelphia, PA, USA*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192404

## 1 Introduction

Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges, or the indexable Web graph, which has roughly 100 billion nodes and trillions of edges. Parallel computing is essential for processing graphs of this size in reasonable time. McSherry *et al.* [44] have shown that

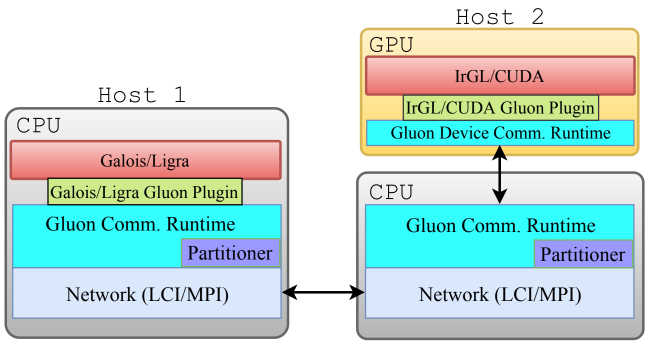


**Figure 1.** Overview of the Gluon Communication Substrate.

shared-memory graph analytics systems like Galois [53] and Ligra [62] process medium-scale graphs efficiently, but these systems cannot be used for graphs with billions of nodes and edges because main memory is limited to a few terabytes even on large servers.

One solution is to use clusters, which can provide petabytes of storage for in-memory processing of large graphs. Graphs are partitioned between the machines, and communication between partitions is implemented using a substrate like MPI. Existing systems in this space such as PowerGraph [23], PowerLyra [17], and Gemini [75] have taken one of two approaches regarding communication. Some systems build optimized communication libraries for a single graph partitioning strategy; for example, Gemini supports only edge-cut partitioning. However, the best-performing graph partitioning strategy depends on the algorithm, input graph, and number of hosts, so this approach is not sufficiently flexible. An alternative approach taken in systems like PowerGraph and PowerLyra is to support vertex-cut graph partitioning. Although vertex-cuts are general, communication is implemented using the generic gather-apply-scatter model [23], which does not take advantage of partitioning invariants to optimize communication. Since performance on large clusters is limited by communication overhead, a key challenge is to optimize communication while supporting heterogeneous partitioning policies. This is explained further in Section 2 using a simple example.

Another limitation of existing systems is that they are integrated solutions that come with their own programming models, runtime systems, and communication runtimes, which makes it difficult to reuse infrastructure to build new systems. For example, all existing GPU graph analytics systems such as Gunrock [56, 69], Groute [8], and IrGL [55] are limited to a single node, and there is no way to reuse infrastructure from existing distributed graph analytics systems to build GPU-based distributed graph analytics systems from these single-node systems.

*This paper introduces a new approach to building distributed-memory graph analytics systems that exploits heterogeneity in programming models, partitioning policies, and processor types (CPU and GPU).* The key to this approach is Gluon, a communication-optimizing substrate.

Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run efficiently on heterogeneous clusters by partitioning the input graph using a policy that can be chosen at runtime and by optimizing communication for that policy.

To demonstrate Gluon's support for *heterogeneous programming models*, we integrated Gluon with the Galois [53] and Ligra [62] shared-memory systems to build distributed graph analytics systems that we call *D-Galois* and *D-Ligra*, respectively. To demonstrate Gluon's support for *processor heterogeneity*, we integrated Gluon with IrGL [55], a single-GPU graph analytics system, to build *D-IrGL*, the first cluster-based multi-GPU graph analytics system.

Figure 1 illustrates a distributed, heterogeneous graph analytics system that can be constructed with Gluon: there is a CPU host running Galois or Ligra, and a GPU, connected to another CPU, running IrGL.

Another contribution of Gluon is that it incorporates novel communication optimizations that *exploit structural and temporal invariants of graph partitions to optimize communication.*

- *Exploiting structural invariants*: We show how general graph partitioning strategies can be supported in distributed-memory graph analytics systems while still exploiting structural invariants of a given graph partitioning policy to optimize communication (Section 3).
- *Exploiting temporal invariance*: The partitioning of the graph does not change during the computation. We show how this *temporal invariance* can be exploited to reduce both the overhead and the volume of communication compared to existing systems (Section 4).

Our evaluation in Section 5 shows that the CPU-only systems D-Ligra and D-Galois are faster than Gemini [75], the state-of-the-art distributed-memory CPU-only graph analytics system, on CPU clusters with up to 256 hosts and roughly 70,000 threads. The geomean speedup of D-Galois over Gemini is ~3.9× even though D-Galois, unlike Gemini, is not a monolithic system. We also show that D-IrGL is effective as a multi-node, multi-GPU graph analytics system on multi-GPU clusters with up to 64 GPUs, yielding a geomean speedup of ~4.9× over Gemini. Finally, we demonstrate that our communication optimizations result in a ~2.6× geomean improvement in running time compared to a baseline in which these optimizations are turned off.

## 2 Overview of Gluon

Gluon can be interfaced with any shared-memory graph analytics system that supports the vertex programming model, which is described briefly in Section 2.1. Section 2.2 gives a high-level overview of how Gluon enables such systems to run on distributed-memory machines. Section 2.3 describes opportunities for optimizing communication compared to the baseline gather-apply-scatter model [23] of synchronization.

### 2.1 Vertex Programs

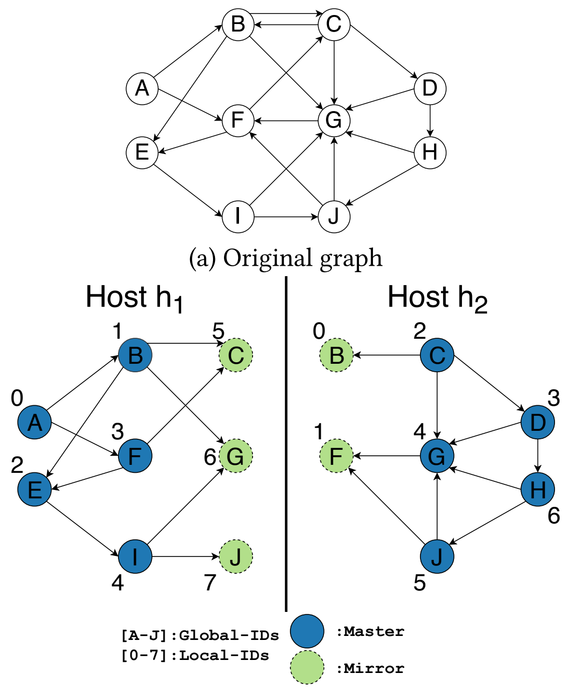
In the graph analytics applications considered in this paper, each node $v$ has a label $l(v)$ that is initialized at the beginning of the algorithm and updated during the execution of the algorithm until a global quiescence condition is reached. In some problems, edges also have labels, and the label of an edge $(v, w)$ is denoted by $weight(v, w)$. We use the single-source shortest-path (sssp) problem to illustrate concepts. Vertex programs for this problem initialize the label of the source node to zero and the label of every other node to a large positive number. At the end of the computation, the label of each node is the distance of the shortest path to that node from the source node.

Updates to node labels are performed by applying a computation rule called an *operator* [58] to nodes in the graph. A *push-style* operator uses the label of a node to conditionally update labels of its immediate neighbors, while a *pull-style* operator reads the labels of the immediate neighbors and conditionally updates the label of the node where the operator is applied. For the sssp problem, the operator is known as the relaxation operator. A push-style relaxation operator sets the label of a neighbor $w$ of the node $v$ to the value $l(v) + weight(v, w)$ if $l(w)$ is larger than this value [19]. Shared-memory systems like Galois repeatedly apply operators to graph nodes until global quiescence is reached.

### 2.2 Distributed-Memory Execution

Distributed-memory graph analytics is more complicated since it is necessary to perform both computation and communication. The graph is partitioned between hosts at the start of the computation. Execution is done in rounds: in each round, a host applies the operator to graph nodes in its own partition and then participates in a global communication phase in which it exchanges information about labels of nodes at partition boundaries with other hosts. Since fine-grain communication is very expensive on current systems, execution models with coarse-grain communication, such as bulk-synchronous parallel (BSP) execution, are preferred [67].

To understand the issues that arise in coupling shared-memory systems on different (possibly heterogeneous) hosts to create a distributed-memory system for graph analytics, consider Figure 2(a), which shows a directed graph with ten



(a) Original graph

(b) Partitioned graph

**Figure 2.** An example of partitioning a graph for two hosts.

nodes labeled A through H (the *global-IDs* of nodes). There are two hosts $h_1$ and $h_2$, and the graph is partitioned between them. Figure 2(b) shows the result of applying an *Outgoing Edge-Cut* (OEC) partitioning (described in Section 3.1): nodes {A,B,E,F,I} have been assigned to host $h_1$ and the other nodes have been assigned to host $h_2$. Each host creates a *proxy node* for the nodes assigned to it, and that proxy node is said to be a *master*: it keeps the canonical value of the node.

Some edges, such as edge (B,G) in Figure 2(a), connect nodes assigned to different hosts. For these edges, OEC partitioning creates a *mirror node* for the destination node (in the example, node G) on the host that owns the source node (in the example, host $h_1$), and it creates an edge on $h_1$ between the proxy nodes for B and G. The following invariants hold in the partitioned graph of Figure 2(b).

a) Every node N in the input graph is assigned to one host $h_i$. Proxy nodes for N are created on this host and possibly other hosts. The proxy node on $h_i$ is called the master proxy for N, and the others are designated mirror proxies.
b) In the partitioned graph, all edges connect proxy nodes on the same host.

Consider a push-style sssp computation. Because of invariant (b), the application program running on each host is oblivious to the existence of other partitions and hosts, and it can execute its sssp code on its partition independently of other hosts using any shared-memory graph analytics system (*this is the key insight that allows us to interface such systems with a common communication-optimizing substrate*). Node G in the graph of Figure 2 has proxies on both hosts, and both proxies have incoming edges, so the labels on both proxies may be updated and read independently by the application programs running on the two hosts. Since mirror nodes implement a form of software-controlled caching, it is necessary to reconcile the labels on proxies at some point. In BSP-style synchronization, collective communication is performed among all hosts to reconcile the labels on all proxies. At the end of synchronization, computation resumes at each host, which is again agnostic of other partitions and hosts.

In the sssp example, the label of the mirror node for G on host $h_1$ can be transmitted to $h_2$, and the label of the master node for G on $h_2$ can be updated to the minimum of the two labels. In general, a node may have several mirror proxies on different hosts. If so, the values on the mirrors can be communicated to the master, which reduces them and updates its label to the resulting value. This value can then be broadcast to the mirror nodes, which update their labels to this value. This general approach is called *gather-apply-scatter* in the literature [23].

### 2.3 Opportunities for Communication Optimization

Communication is the performance bottleneck in graph analytics applications [70], so communication optimizations are essential to improving performance.

The first set of optimizations exploit structural invariants of partitions to reduce the amount of communication compared to the default gather-apply-scatter implementation. In Figure 2(b), we see that mirror nodes do not have outgoing edges; this is an invariant of the OEC partitioning (explained in Section 3). This means that a push-style operator applied to a mirror node is a no-op and that the label on the mirror node is never read during the computation phase. Therefore, the volume of communication can be reduced in half by just resetting the labels of mirror nodes locally instead of updating them to the master's value during the communication phase, which obviates the need to communicate values from masters to the mirrors. The value to reset the mirror nodes to depends on the operator. For example, for sssp, keeping labels of mirror nodes unchanged is sufficient whereas for push-style pagerank, the labels are reset to 0. In Section 3, we describe a number of important partitioning strategies, and we show how Gluon can be used to exploit their structural invariants to optimize communication.

The second set of optimizations, described in Section 4, reduces the memory and communication overhead by exploiting the temporal invariance of graph partitions. Once the graph has been partitioned, each host stores its portion of the graph using a representation of its choosing. Proxies assigned to a given host are given *local-IDs*, and the graph structure is usually stored in Compressed-Sparse-Row (CSR) format, which permits the proxies assigned to a given host to be stored contiguously in memory regardless of their global-IDs. Figure 2(b) shows an assignment of local-IDs (numbers in the figure) to the proxies.

Since local-IDs are used only in intra-host computation and have no meaning outside that host, communication between a master and its mirrors on different hosts requires reference to their common global-ID. In current systems, each host maintains a map between local-IDs and global-IDs. To communicate the label of a proxy on host $h_1$ to the corresponding proxy on host $h_2$, (i) the local-ID of the proxy on $h_1$ is translated to the global-ID, (ii) the global-ID and node label are communicated to host $h_2$, and (iii) the global-ID is translated to the corresponding local-ID on $h_2$. This approach has two overheads: conversion between global-IDs and local-IDs and communication of global-IDs with labels.

Gluon implements an important optimization called *memoization of address translation* (Section 4.1), which obviates the need for the translation and for sending global-IDs. A second optimization (Section 4.2) tracks updates to proxies and avoids sending proxy values that have not changed since the previous round. While this optimization is implemented in other systems such as PowerGraph and Gemini, these systems send global-IDs along with proxy values. Implementing this optimization in a system like Gluon is more challenging for two reasons: it is not an integrated computation/communication solution, and it does not send global-IDs with proxy values. Section 4.2 describes how we address this problem.

## 3 Exploiting Structural Invariants to Optimize Communication

Section 3.1 describes partitioning strategies implemented in Gluon. Section 3.2 describes how communication can be optimized by using the structural invariants of these strategies. Section 3.3 describes the Gluon API calls that permit these optimized communication patterns to be plugged in with existing shared-memory graph analytical systems.

### 3.1 Partitioning Strategies

The partitioning strategies implemented by Gluon can be presented in a unified way as follows. The edges of the graph are distributed between hosts using some policy. If a host is assigned an edge $(N_1, N_2)$, proxies are created for nodes $N_1$ and $N_2$ and are connected by an edge on that host. If the edges connected to a given node end up on several hosts, that node has several proxies in the partitioned graph. One proxy is designated the master for all proxies of that node, and the others are designated as mirrors. The master is responsible for holding and communicating the canonical value of the node during synchronization.

Partitioning policies can interact with the structure of the operator in the sense that some partitioning policies may not

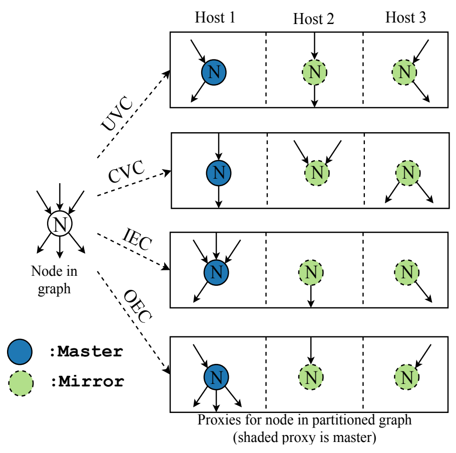

Host 1
Host 2
Host 3
UVC
CVC
IEC
OEC
N
Node in graph
:Master
:Mirror
Proxies for node in partitioned graph (shaded proxy is master)

**Figure 3.** Different partitioning strategies.

be legal for certain operators. To understand this interaction, it is useful to classify various policies (or heuristics) for graph partitioning into four categories or strategies. These are described below and illustrated in Figure 3. In this figure, a node $N$ in the graph has three proxies on different hosts; the shaded proxy is the master.

- *Unconstrained Vertex-Cut* (*UVC*) can assign both the outgoing and incoming edges of a node to different hosts.
- *Cartesian Vertex-Cut* (*CVC*) is a constrained vertex-cut: only the master can have both incoming and outgoing edges, while mirror proxies can have either incoming or outgoing edges but not both.
- *Incoming Edge-Cut* (*IEC*) is more constrained: incoming edges are assigned only to the master while outgoing edges of a node are partitioned among hosts.
- *Outgoing Edge-Cut* (*OEC*) is the mirror image of IEC: outgoing edges are assigned only to the master while incoming edges are partitioned among hosts.

Some of these partitioning strategies can be used only if the operator has a special structure. For a pull-style operator, UVC, CVC, or OEC can be used only if the update made by the operator to the active node label is a reduction; otherwise, IEC must be used since all the incoming edges required for local computation are present at the master. For a push-style operator, UVC, CVC, or IEC can be used only if the node pushes the same value along its outgoing edges and uses a reduction to combine the values pushed to it on its incoming edges; otherwise, OEC must be used since master nodes have all the outgoing edges required for the computation. The benchmarks studied in this paper satisfy all these conditions, so any partitioning strategy can be used for them.

### 3.2 Partitioning Invariants and Communication

Each partitioning strategy requires a different pattern of optimized communication to synchronize proxies, but they can be composed from two basic patterns supported by a thin API. The first is a *reduce* pattern in which values at mirror nodes are communicated to the host that owns the master and combined on that host using a reduction operation. The resulting value is written to the master. The second is a *broadcast* pattern in which data at the master is broadcast to the mirrors. For some partitioning strategies, only a subset of mirrors may be involved in the communication.

Consider push-style operators that read the label of the active node and write to the labels of outgoing neighbors or pull-style operators that read the labels of incoming neighbors and write to the label of the active node. In these cases, the data flows from the source to the destination of an edge. We discuss only the synchronization patterns for this scenario in this section; synchronization patterns for other cases are complementary and are not discussed further.

Since the data flows from the source to the destination of an edge, a proxy's label is only read if it is the source node of an edge and only written if it is the destination node of an edge (a reduction is considered to be a write). Consequently, there are two invariants during execution:

(1) If a proxy node has no outgoing edges, its label will not be read during the computation phase.
(2) If a proxy node has no incoming edges, its label will not be written during the computation phase.

These invariants can be used to determine the communication patterns required for the different partitioning strategies.

- *UVC*: Since the master and mirrors of a node can have outgoing as well as incoming edges, any proxy can be written during the computation phase. At the end of the round, the labels of the mirrors are communicated to the master and combined to produce a final value. The value is written to the master and broadcast to the mirror nodes. Therefore, both reduce and broadcast are required. This is the default gather-apply-scatter pattern used in systems like PowerGraph [23].
- *CVC*: Only the master can have both incoming and outgoing edges; mirrors can have either incoming or outgoing edges but not both. Consequently, mirrors either read from the label or write to the label but not both. At the end of the round, the set of mirrors that have only incoming edges communicate their values to the master to produce the final value. The master then broadcasts the value to the set of mirrors that have only outgoing edges. Like UVC, CVC requires both reduce and broadcast synchronization patterns, but each pattern uses only a particular subset of mirror nodes rather than all mirror nodes. This leads to better performance at scale for many programs.
- *IEC*: Only the master has incoming edges, so only its label can be updated during the computation. The master communicates this updated value to the mirrors for

```
1  struct SSSP { /* SSSP edgemap functor */
2    unsigned int* dist;
3    SSSP(unsigned int* _dist) : dist(_dist) { }
4    bool update(unsigned s, unsigned d, int edgeLen) {
5      unsigned int new_dist = dist[s] + edgeLen;
6      if (dist[d] > new_dist) {
7        dist[d] = new_dist;
8        return true;
9      }
10     return false;
11   }
12   ... /* other Ligra functor functions */
13 };
14 void Compute(...) { /* Main computation loop */
15   ... /* setup initial local-frontier */
16   do {
17     edgeMap(LigraGraph, LocalFrontier, SSSP(dist),
        ...);
18     Gluon.sync<WriteAtDestination, ReadAtSource,
        ReduceDist, BroadcastDist>(LocalFrontier);
19     /* update local-frontier for next iteration */
20   } while (LocalFrontier is non-zero on any host);
21 }
```

**Figure 4.** Ligra plugged in with Gluon: sssp (D-Ligra).

```
struct ReduceDist { /* Reduce struct */
  static unsigned extract(unsigned localNodeID) {
    return dist[localNodeID];
  }
  static bool reduce(unsigned localNodeID, unsigned y)
  {
    if (y < dist[localNodeID]) { /* min operation */
      dist[localNodeID] = y;
      return true;
    }
    return false;
  }
  static void reset(unsigned localNodeID) {
    /* no-op */
  }
};
struct BroadcastDist { /* Broadcast struct */
  static unsigned extract(unsigned localNodeID) {
    return dist[localNodeID];
  }
  static void set(unsigned localNodeID, unsigned y) {
    dist[localNodeID] = y;
  }
};
```

**Figure 5.** Gluon synchronization structures: sssp (D-Ligra).

the next round. Therefore, only the broadcast synchronization pattern is required. This is the *halo exchange* pattern used in distributed-memory finite-difference codes.

- *OEC*: Only the master of a node has outgoing edges. At the end of the round, the values pushed to the mirrors are combined to produce the final result on the master, and the values on the mirrors can be reset to the (generalized) zero of the reduction operation for the next round. Therefore, only the reduce synchronization pattern is required.

## 3.3 Synchronization API

To interface programs with Gluon, a (blocking) synchronization call is inserted between successive parallel rounds in the program (*e.g.*, after edgeMap() in Ligra, do_all() in Galois, Kernel in IrGL). Figure 4 shows how Gluon can be used in an sssp application in Ligra to implement communication-optimized distributed sssp (D-Ligra). The synchronization call to the Gluon substrate shown in line 18 passes the reduce and broadcast API functions shown in Figure 5 to Gluon. Synchronization is field-sensitive: therefore, synchronization structures similar to the ones shown in Figure 5 are used for each label updated by the program. Depending on the partitioning strategy, Gluon uses reduce, broadcast, or both.

The functions in the reduce and broadcast structures specify how to access data that must be synchronized (there are also bulk-variants for GPUs). Broadcast has two main functions:

- *Extract*: Masters call this function to extract their node field values from the local graph to broadcast them to mirrors.
- *Set*: Mirrors call this function to update their node field values in the local graph to the canonical value received from masters.

Reduce has three main functions:

- *Extract*: Mirrors call this function to extract their node field values from the local graph to communicate them to the master.
- *Reduce*: Masters call this function to combine the partial values received from the mirrors to their node field values in the local graph.
- *Reset*: Mirrors call this function to reset their node field values in the local graph to the identity-value of the reduction operation for the next round.

Note that the application code does not depend on the particular partitioning strategy used by the programmer. Instead, Gluon composes the optimized communication pattern from the information in the sync call and its knowledge of the communication requirements of the particular partitioning strategy requested by the programmer. This permits programmers to explore a variety of partitioning strategies just by changing command-line flags, which permits autotuning.

The approach described in this section decouples the computation from communication, which enables the computation to run on any device or engine using any scheduling strategy. For each compute engine (Ligra, Galois, and IrGL), the broadcast and reduction structures can be supported through application-agnostic preprocessor templates. Each application only needs to identify the field(s) to synchronize, the reduction operation(s), and the point(s) at which to synchronize. This can be identified by statically analyzing the operator (*e.g.*, edgeMap() in Ligra, do_all() in Galois, Kernel in IrGL) and inserting the required sync call (we have implemented this in a compiler for Galois). Communication is automatically specialized for the partitioning policy specified at runtime.

## 4 Exploiting Temporal Invariance to Optimize Communication

The optimizations in Section 3 exploit properties of the operator and partitioning strategy to reduce the volume of communication for synchronizing the proxies of a *single* node in the graph. In general, there are many millions of nodes whose proxies need to be synchronized in each round. In this section, we describe two optimizations for reducing overheads when performing this mass communication. The first optimization, described in Section 4.1, permits proxy values to be exchanged between hosts without sending the global-IDs of nodes. The second optimization, described in Section 4.2, tracks updates to proxies and avoids sending proxy values that have not changed since the previous round.

### 4.1 Memoization of Address Translation

As described in Section 3, synchronizing proxies requires sending values from mirrors to masters or vice versa. Mirrors and masters are on different hosts, so the communication needed for this is handled in current systems by sending node global-IDs along with values. Consider Figure 2(b): if host $h_2$ needs to send host $h_1$ the labels on the mirrors for nodes B and F in its local memory, it sends the global-IDs B and F along with these values. At the receiving host, these global-IDs are translated into local-IDs (in this case, 1 and 3), and the received values are assimilated into the appropriate places in the local memory of that host.

This approach has both time and space overheads: communication volume and time increase because the global-IDs are sent along with values, and computational overhead increases because of the translation between global and local-IDs. To reduce these overheads, Gluon implements a key optimization called *memoization of address translation* that eliminates the sending of global-IDs as well as the translation between global and local-IDs.

In Gluon, as in existing distributed-memory graph analytics systems, graph partitions and the assignment of partitions to hosts do not change during the computation[1]. Gluon exploits this temporal invariance of partitions as follows. Before the computation begins, Gluon establishes an agreement between pairs of hosts $(h_i, h_j)$ that determines which proxies on $h_i$ will send values to $h_j$ and the order in which these values will be packed in the message.

This high-level idea is implemented as follows. We use Figure 6, which shows the memoization of address translation for the partitions in Figure 2, to explain the key ideas.

[1]If the graph is re-partitioned, then memoization can be done soon after partitioning to amortize the communication costs until the next re-partitioning.

In Gluon, each host reads from disk a subset of edges assigned to it and receives from other hosts the rest of the edges assigned to it. The end-points of these edges are specified using global-IDs. When building the in-memory (CSR) representation of its local portion of the graph, each host maintains a map to translate the global-IDs of its proxies to their local-IDs and a vector to translate the local-IDs to global-IDs.

After this, each host informs every other host about the global-IDs of its mirror nodes whose masters are owned by that other host. In the running example, host $h_1$ informs $h_2$ that it has mirrors for nodes with global-IDs {C,G,J}, whose masters are on $h_2$. It does this by constructing the *mirrors* array shown in Figure 6 and sending it to $h_2$. This exchange of mirrors provides the *masters* array shown in the figure. After the exchange, the global-IDs in the *masters* and *mirrors* arrays are translated to their local-IDs. This is the only point where the translation is performed unless the user code explicitly uses the global-ID of a node during computation (e.g., to set the source node in sssp).

During the execution of the algorithm, communication is either from masters to mirrors (during broadcast) or from mirrors to masters (during reduce). Depending on whether it is broadcast or reduce, the corresponding masters or mirrors array (respectively) of local IDs is used to determine what values to send to that host. Once the values are received, the corresponding mirrors or masters array (respectively) is used to update the proxies. Since the order of proxy values in the array has been established during memoization and the messages sent by the runtime respect this ordering, there is no need for address translation.

In addition to reducing communication and address translation overheads, an important benefit of this optimization is that it enables Gluon to leverage existing shared-memory frameworks like Galois and Ligra out-of-the-box. Moreover, systems for other architectures like GPUs need not maintain memory-intensive address translation structures, thereby enabling Gluon to directly leverage GPU frameworks like IrGL.

### 4.2 Encoding of Metadata for Updated Values

In BSP-style execution of graph analytics algorithms, each round usually updates labels of only a small subset of graph nodes. For example, in the first round of bfs, only the neighbors of the source node have their labels updated. An important optimization is to avoid communicating the labels of nodes that have not been updated in a given round. If global-IDs are sent along with label values as is done in existing systems, this optimization is easy to implement [17, 23, 29, 75]. If, however, the memoization optimization described in Section 4.1 is used, it is not clear how to send only the updated values in a round since receivers will not know which nodes these values belong to.

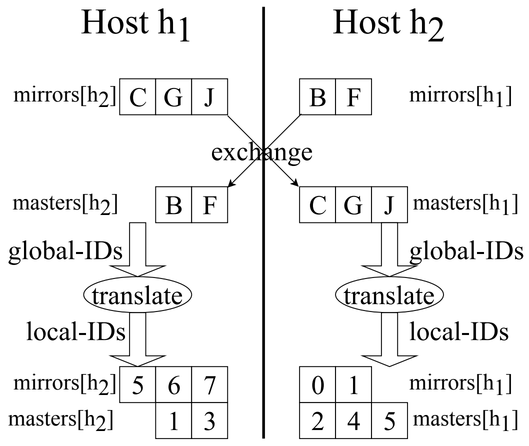


**Figure 6.** Memoization of address translation for the partitions in Figure 2.

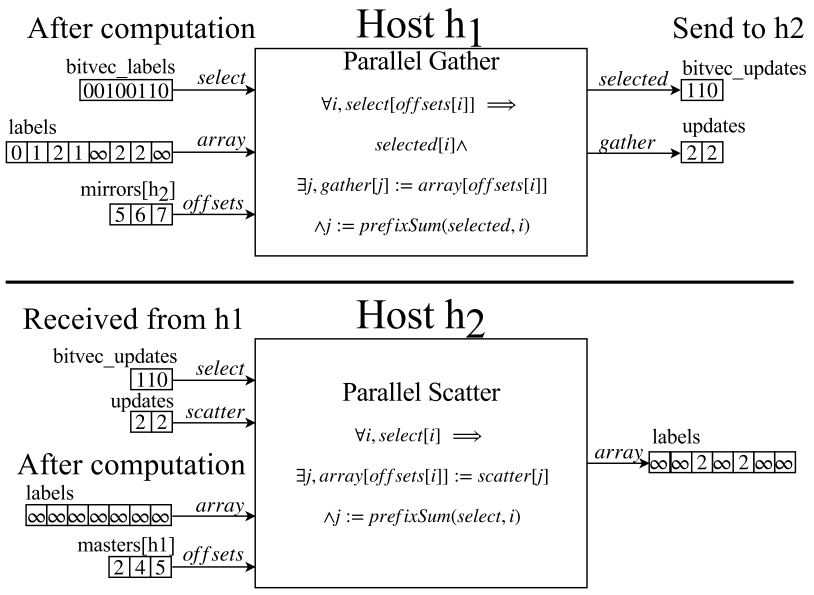


**Figure 7.** Communication from host $h_1$ to $h_2$ after second round of BFS algorithm with source A on the OEC partitions in Figure 2.

One way to track the node labels that have changed is to keep a shadow copy of it in the communication runtime and compare the original against it to determine if the label has changed. To be more space efficient, Gluon requires the shared-memory runtime to provide a field-specific bit-vector that indicates which nodes' labels have changed. This is passed to the synchronization call to Gluon (e.g., in Figure 4, `LocalFrontier` in line 18).

To illustrate this idea, we consider a level-by-level BFS algorithm on the partitions in Figure 2 with source $A$. Since these are OEC partitions, the mirrors need to be reduced to the master after each round (described in Section 3.2). In the first round, host $h_1$ updates the labels of $B$ and $F$ to 1. There is nothing to communicate since those nodes are not shared with $h_2$. After the second round, $h_1$ updates $C$, $G$, and $E$ to 2. The updates to $C$ and $G$ need to be synchronized with $h_2$.

In the top left of Figure 7, host $h_1$ has the updated labels with its bit-vector indicating which local labels changed in the second round and the *mirrors* of $h_2$ from memoization (Figure 6). Instead of gathering all mirrors, only the mirrors that are set in the bit-vector are selected and gathered. Out of 5, 6, and 7 ($C$, $G$, and $J$, respectively), only 5 and 6 are set in the bit-vector. This yields the *updates* array of 2 and 2 shown in the top right. In addition, a bit-vector is determined which indicates which of the mirrors were selected. The *bitvec_updates* shows that mirror 0 and 1 (0-indexed) were selected from the mirrors. This bit-vector is sent along with the updates to $h_2$.

In the bottom left of Figure 7, host $h_2$ has the labels after the second round of computation and the *masters* of $h_1$ from memoization (Figure 6). It also has the bit-vector and the updates received from $h_1$. The bit-vector is used to select the masters, and the updates are scattered to the labels of those masters. Since 0 and 1 indices are set in the bit-vector, local-IDs 2 and 4 are selected, and their labels are updated with 2 and 2 respectively, yielding the updated labels shown on the bottom right.

When updates are very dense, sending all the labels of the mirrors or masters can obviate the need for the bit-vector metadata since most labels will be updated. To illustrate for the current running example, an array of 22 ∞ would be sent to $h_2$, and $h_2$, using its *masters* array, can correctly scatter them to the $C$, $G$, and $J$. In cases when the updates are very sparse, sending just indices of the selected mirrors or masters instead of the bit-vector can further reduce the size of the metadata. If indices are sent for the example considered, an indices array of 0 and 1 is sent instead of the bit-vector 110. The indices are then used by the receiver to select the masters.

Gluon has the different modes described above to encode the metadata compactly. Simple rules to select the mode are as follows:

- When the updates are dense, do not send any bit-vector metadata, but send labels of all mirrors or masters.
- When the updates are sparse, send bit-vector metadata along with updated values.
- When the updates are very sparse, send the indices along with the updated values.
- When there are no updates, send an empty message.

The number of bits set in the bit-vector is used to determine which mode yields the smallest message size. A byte in the sent message indicates which mode was selected. Neither the

**Table 1.** Inputs and their key properties.

| | rmat26 | twitter40 | rmat28 | kron30 | clueweb12 | wdc12 |
|---|---|---|---|---|---|---|
| $\lvert V\rvert$ | 67M | 41.6M | 268M | 1073M | 978M | 3,563M |
| $\lvert E\rvert$ | 1,074M | 1,468M | 4,295M | 10,791M | 42,574M | 128,736M |
| $\lvert E\rvert/\lvert V\rvert$ | 16 | 35 | 16 | 16 | 44 | 36 |
| max $D_{out}$ | 238M | 2.99M | 4M | 3.2M | 7,447 | 55,931 |
| max $D_{in}$ | 18,211 | 0.77M | 0.3M | 3.2M | 75M | 95M |

rules nor the modes are exhaustive. Other compression or encoding techniques could be used to represent the bit-vector as long as they are deterministic (and sufficiently parallel).

## 5 Experimental Results

Gluon can use either MPI or LCI [20] for message transport between hosts, as shown in Figure 1. We use LCI in our evaluation[2]. To evaluate Gluon, we interfaced it with the Ligra [62], Galois [53], and IrGL [55] shared-memory graph analytics engines to build three distributed-memory graph analytics systems that we call D-Ligra, D-Galois[3], and D-IrGL respectively. We compared the performance of these systems with that of Gunrock [56], the state-of-the-art single-node multi-GPU graph analytics system, and Gemini [75], the state-of-the-art distributed-memory CPU-only graph analytics system (the Gemini paper shows that their system performs significantly better than other systems in this space such as PowerLyra [17], PowerGraph [23], and GraphX [72]).

We describe the benchmarks and experimental platforms (Section 5.1), graph partitioning times (Section 5.2), the performance of all systems at scale (Section 5.3), experimental studies of the CPU-only systems (Sections 5.4), and experimental studies of the GPU-only systems (Section 5.5). Section 5.6 gives a detailed breakdown of the performance impact of Gluon's communication optimizations.

### 5.1 Experimental Setup, Benchmarks, and Input Graphs

All CPU experiments were conducted on the Stampede KNL Cluster (Stampede2) [64] at the Texas Advanced Computing Center [6], a distributed cluster connected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). Each node has a KNL processor with 68 1.4 GHz cores running four hardware threads per core, 96GB of DDR4 RAM, and 16GB of MC-DRAM. We used up to 256 CPU hosts. Since each host has 272 hardware threads, this allowed us to use up to 69632 threads. All code was compiled using g++ 7.1.0.

GPU experiments were done on the Bridges [54] supercomputer at the Pittsburgh Supercomputing Center [5, 65], another distributed cluster connected through Intel Omni-Path Architecture. Each machine has 2 Intel Haswell CPUs with 14 cores each and 128 GB DDR4 RAM, and each is connected to 2 NVIDIA Tesla K80 dual-GPUs (4 GPUs in total with) with 24 GB of DDR5 memory (12 GB per GPU). Each GPU host uses 7 cores and 1 GPU (4 GPUs share a single physical node). We used up to 64 GPUs. All code was compiled using g++ 6.3.0 and CUDA 9.0.

Our evaluation uses programs for breadth-first search (bfs), connected components (cc), pagerank (pr), and single-source shortest path (sssp). In D-Galois and D-IrGL, we implemented a pull-style algorithm for pagerank and push-style data-driven algorithms for the rest. Both push-style and pull-style implementations are available in D-Ligra due to Ligra's direction optimization. The source nodes for bfs and sssp are the maximum out-degree node. The tolerance for pr is $10^{-9}$ for rmat26 and $10^{-6}$ for the other inputs. *We run pr for up to 100 iterations; all the other benchmarks are run until convergence.* Results presented are for a mean of 3 runs.

Table 1 shows the properties of the six input graphs we used in our studies. Three of them are real-world web-crawls: the web data commons hyperlink graph [47, 48], wdc12, is the largest publicly available dataset; clueweb12 [9, 10, 59] is another large publicly available dataset; twitter40 [36] is a smaller dataset. rmat26, rmat28, and kron30 are randomized synthetically generated scale-free graphs using the rmat [16] and kron [40] generators (we used weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]).

### 5.2 Graph Partitioning Policies and Times

We implemented the four high-level partitioning strategies described in Section 3.1, using particular policies to assign edges to hosts for each one. For OEC and IEC, we implemented a chunk-based edge-cut [75] that partitions the node into contiguous blocks while trying to balance outgoing and incoming edges respectively. For CVC, we implemented a 2D graph partitioning policy [11]. For UVC, we implemented a hybrid vertex-cut (HVC) [17]. While Gluon supports a variety of partitioning policies, evaluating different partitioning policies in Gluon is not the focus of this work, so we experimentally determined good partitioning policies for the Gluon-based systems. For bfs, pr, and sssp using D-IrGL on clueweb12, we used the OEC policy. In all other cases for D-Ligra, D-Galois, and D-IrGL, unless otherwise noted, we used the CVC policy since it performs well at scale. Note that Gemini and Gunrock support only outgoing edge-cuts.

Table 2 shows the time to load the graph from disk, partition it (which uses MPI), and construct the in-memory representation on 32 hosts and 256 hosts for D-Ligra, D-Galois, and Gemini. For comparison, we also present the time to load and construct small graphs on a single host for Ligra, Galois, and Gemini. Partitioning the graph on more hosts may increase inter-host communication, but it also increases the total disk bandwidth available, so the total graph construction time on 256 hosts for rmat28 is *better* than the total graph construction time on 1 host for all systems. Similarly, the graph construction time on 256 hosts is better than that

[2]Dang et al. [20] show the benefits of LCI over MPI for graph analytics.
[3]The Abelian system used in [20] is another name for D-Galois.

**Table 2.** Graph construction time (sec): time to load the graph from disk, partition it, and construct in-memory representation.

| 1 host | rmat26 | twitter40 | rmat28 |
|---|---|---|---|
| Ligra | 271.6 | 158.3 | 396.9 |
| Galois | **64.9** | **51.8** | **123.9** |
| Gemini | 854.3 | 893.5 | 3084.7 |

| 32 hosts | rmat28 | kron30 | clueweb12 |
|---|---|---|---|
| D-Ligra | 70.6 | 257.4 | 728.6 |
| D-Galois | **68.6** | **244.4** | **600.8** |
| Gemini | 328.0 | 1217.4 | 1539.0 |

| 256 hosts | rmat28 | kron30 | clueweb12 | wdc12 |
|---|---|---|---|---|
| D-Ligra | 69.4 | 235.8 | 470.5 | 1515.9 |
| D-Galois | **65.5** | **225.7** | **396.2** | **1345.0** |
| Gemini | 231.0 | 921.8 | 1247.7 | X |

**Table 3.** Fastest execution time (sec) of all systems using the best-performing number of hosts: distributed CPUs on Stampede and distributed GPUs on Bridges (number of hosts in parenthesis; "-" means out-of-memory; "X" means crash).

| Bench- | Input | CPUs | | | GPUs |
|---|---|---|---|---|---|
| mark | | D-Ligra | D-Galois | Gemini | D-IrGL |
| bfs | rmat28 | 1.0 (128) | **0.8 (256)** | 2.3 (8) | **0.5 (64)** |
| | kron30 | 1.6 (256) | **1.4 (256)** | 5.0 (8) | **1.2 (64)** |
| | clueweb12 | 65.3 (256) | **16.7 (256)** | 44.4 (16) | **10.8 (64)** |
| | wdc12 | 1995.3 (64) | **380.8 (256)** | X | - |
| cc | rmat28 | 1.4 (256) | **1.3 (256)** | 6.6 (8) | **1.1 (64)** |
| | kron30 | 2.7 (256) | **2.5 (256)** | 14.6 (16) | **2.5 (64)** |
| | clueweb12 | 52.3 (256) | **8.1 (256)** | 30.2 (16) | **23.8 (64)** |
| | wdc12 | 176.6 (256) | **75.3 (256)** | X | - |
| pr | rmat28 | **19.7 (256)** | 24.0 (256) | 108.4 (8) | **21.6 (64)** |
| | kron30 | **74.2 (256)** | 102.4 (256) | 190.8 (16) | **70.9 (64)** |
| | clueweb12 | 821.1 (256) | **67.0 (256)** | 257.9 (32) | **215.1 (64)** |
| | wdc12 | 663.1 (256) | **158.2 (256)** | X | - |
| sssp | rmat28 | 2.1 (256) | **1.4 (256)** | 6.3 (4) | **1.1 (64)** |
| | kron30 | 3.1 (256) | **2.4 (256)** | 13.9 (8) | **2.3 (64)** |
| | clueweb12 | 112.5 (256) | **28.8 (128)** | 128.3 (32) | **15.8 (64)** |
| | wdc12 | 2985.9 (256) | **574.9 (256)** | X | - |

**Table 4.** Execution time (sec) on a single node of Stampede ("-" means out-of-memory).

| Input | twitter40 | | | | rmat28 | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | bfs | cc | pr | sssp | bfs | cc | pr | sssp |
| Ligra | **0.31** | 2.75 | 175.67 | 2.60 | **0.77** | 17.56 | 542.51 | - |
| D-Ligra | 0.44 | 3.16 | 188.70 | 2.92 | 1.21 | 18.30 | 597.30 | - |
| Galois | 0.68 | 2.73 | **43.47** | 5.55 | 2.54 | 13.20 | **116.50** | 21.42 |
| D-Galois | 1.03 | **1.04** | 86.53 | **1.84** | 4.05 | **7.02** | 326.88 | **5.47** |
| Gemini | 0.85 | 3.96 | 80.23 | 3.78 | 3.44 | 20.34 | 351.65 | 41.77 |

on 32 hosts. D-Ligra and D-Galois use the Gluon partitioner but construct different in-memory representations, so there is a difference in their times. Both systems are faster than Gemini, which uses the simpler edge-cut partitioning policy. On 128 and 256 hosts, the replication factor (average number of proxies per vertex) in Gemini's partitions for different inputs vary from ~4 to 25 while the replication factor in Gluon's partitions (CVC) is smaller and varies from ~2 to 8. This is experimental evidence that Gluon keeps the replication factor relatively low compared to Gemini.

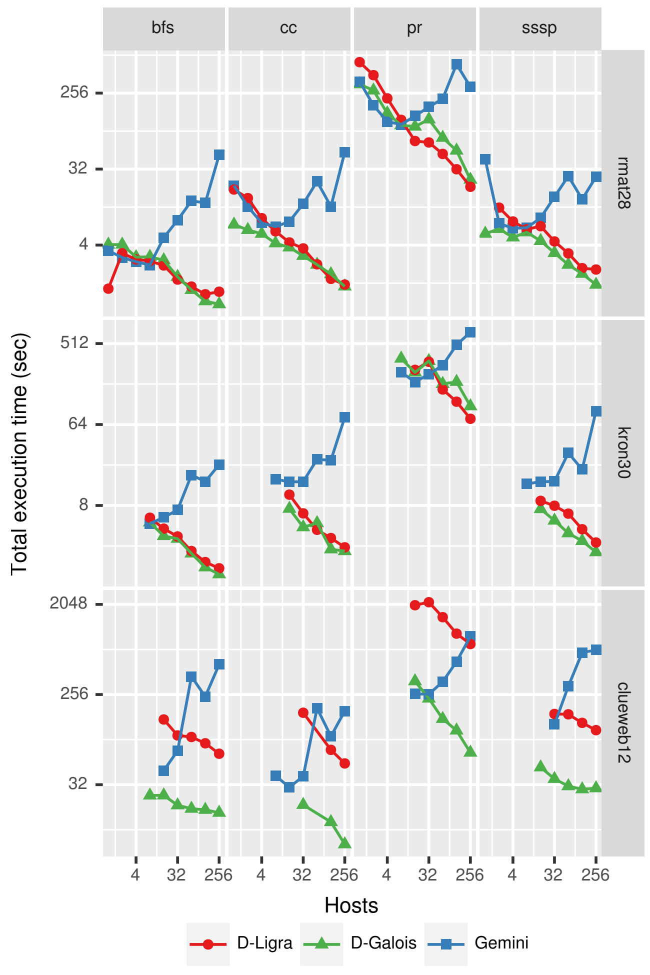
## 5.3 Best Performing Versions

Table 3 shows the execution times of all systems considering their best performance using any configuration or number of hosts across the platforms on all graphs, except the small graphs rmat26 and twitter40 (Gunrock runs out of memory for all other graphs). The configuration or number of hosts that yielded the best execution time is included in parenthesis (it means that the system did not scale beyond that). D-Galois clearly outperforms Gemini, and it can run large graphs like wdc12, which Gemini cannot. D-Ligra does not perform well on very large graphs like clueweb12 and wdc12. For the other graphs, D-Ligra performs better than Gemini. For graphs that fit in 64 GPUs, D-IrGL not only outperforms Gemini but also is generally comparable or better than D-Galois using up to 256 CPUs. We attribute this to the compute power of GPUs and their high memory bandwidth. Comparing the best-performing configurations on different systems, we see that D-Ligra, D-Galois, and D-IrGL give a geomean speedup of ~2×, ~3.9×, and ~4.9× over Gemini, respectively. These results show that the flexibility and support for heterogeneity in Gluon do not come at the expense of performance compared to the state-of-the-art.

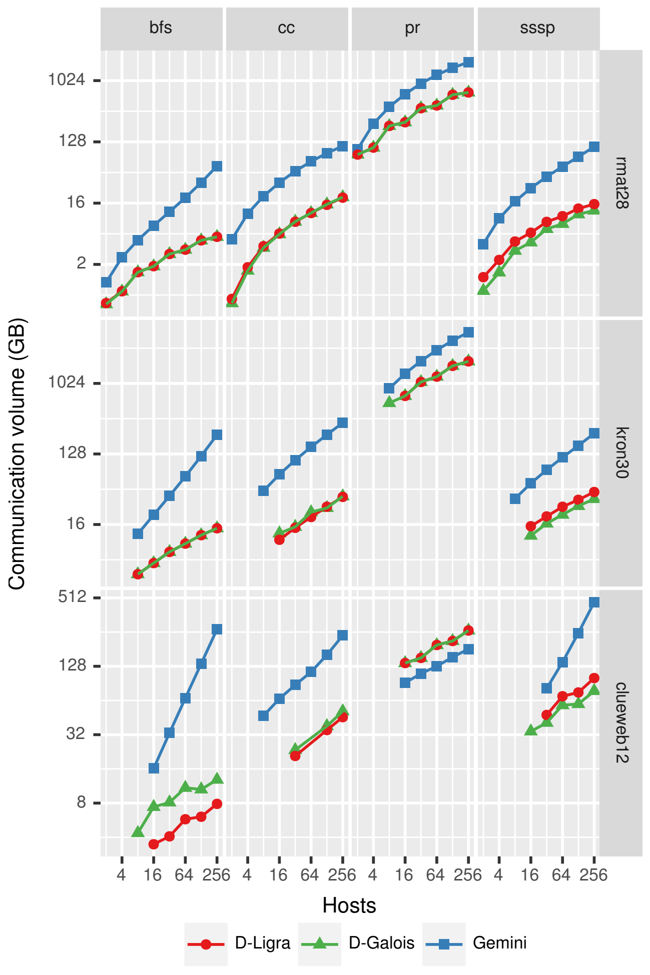
## 5.4 Strong Scaling of Distributed CPU Systems

We first consider the performance of different systems on a *single* host using twitter40 and rmat28, which fit in the memory of one host. The goal of this study is to understand the overheads introduced by Gluon on a single host compared to shared-memory systems like Ligra (March 19, 2017 commit) and Galois (v4.0).

Table 4 shows the single-host results. Note that we used the implementations in the Lonestar [4] benchmark suite (v4.0) for Galois, which may not be vertex programs and may use asynchronous execution. D-Galois uses bulk-synchronous vertex programs as do Ligra and other existing distributed graph analytical systems. Ligra and Gemini use a direction-optimizing algorithm for bfs, so they outperform both Galois and D-Galois for bfs. The algorithms in Lonestar and D-Galois have different trade-offs; e.g., Lonestar cc uses pointer-jumping which is optimized for high-diameter graphs while D-Galois uses label-propagation which is better for low-diameter graphs. The main takeaway from this table is that



(a) Execution time (log-log scale)



(b) Communication volume (log-log scale)

**Figure 8.** Strong scaling of D-Ligra, D-Galois, and Gemini on the Stampede supercomputer (each host is a 68-core KNL node).

D-Galois and D-Ligra are competitive with Lonestar-Galois and Ligra respectively for power-law graphs on one host for all benchmarks, which illustrates that the overheads introduced by the Gluon layer are small.

Figure 8(a) compares the strong scaling of D-Ligra, D-Galois, and Gemini up to 256 hosts[4]. The first high-level point is that for almost all applications, input graphs, and numbers of hosts, D-Galois performs better than Gemini. The second high-level point is that Gemini generally does not scale beyond 16 hosts for any benchmark and input combination. In contrast, D-Ligra and D-Galois scale well up to 256 hosts in most cases. D-Galois on 128 hosts yields a geomean speedup of ~1.7× over itself on 32 hosts.

For the most part, D-Ligra and D-Galois perform similarly for rmat28 and kron30 on all applications and number of hosts. However, their performance for very large graphs are significantly different. D-Ligra performs level-by-level bfs, cc, and sssp in which updates to labels of vertices in the current round are only visible in the next round; in contrast, D-Galois propagates such updates in the same round within the same host (like chaotic relaxation in sssp). Consequently, for these algorithms, D-Ligra has 2 − 4× more rounds and synchronization (implicit) barriers, resulting in much more communication overhead. *These results suggest that for large-scale graph analytics, hybrid solutions that use round-based algorithms across hosts and asynchronous algorithms within hosts might be the best choice.*

Since the execution time of distributed-memory graph analytics applications is dominated by communication time, we measured the bytes sent from one host to another to understand the performance differences between the systems. Figure 8(b) shows the total volume of communication. The main takeaway here is that D-Ligra and D-Galois, which are both based on Gluon, communicate similar volumes of data. Since D-Galois updates vertices in the same round (in an asynchronous manner), it sends more data than D-Ligra for

[4]rmat26 and twitter40 are too small; wdc12 is too large to fit on fewer hosts. Missing point indicates graph loading or partitioning ran out-of-memory.

**Table 5.** Execution time (sec) on a single node of Bridges with 4 K80 GPUs ("-" means out-of-memory).

| Input | rmat26 | | | | twitter40 | | | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **bfs** | **cc** | **pr** | **sssp** | **bfs** | **cc** | **pr** | **sssp** |
| **Gunrock** | - | 1.81 | 51.46 | 1.42 | 0.88 | 1.46 | 37.37 | 2.26 |
| **D-IrGL(OEC)** | 3.61 | 5.72 | 55.72 | 4.13 | 1.03 | 1.57 | 62.81 | 1.99 |
| **D-IrGL(IEC)** | **0.72** | 7.88 | **7.65** | **0.84** | **0.73** | 1.55 | **35.03** | **1.44** |
| **D-IrGL(HVC)** | 0.82 | **1.53** | 8.54 | 0.95 | 1.08 | 1.58 | 44.35 | 2.04 |
| **D-IrGL(CVC)** | 2.11 | 4.22 | 46.91 | 2.24 | 0.87 | **1.39** | 46.86 | 2.32 |

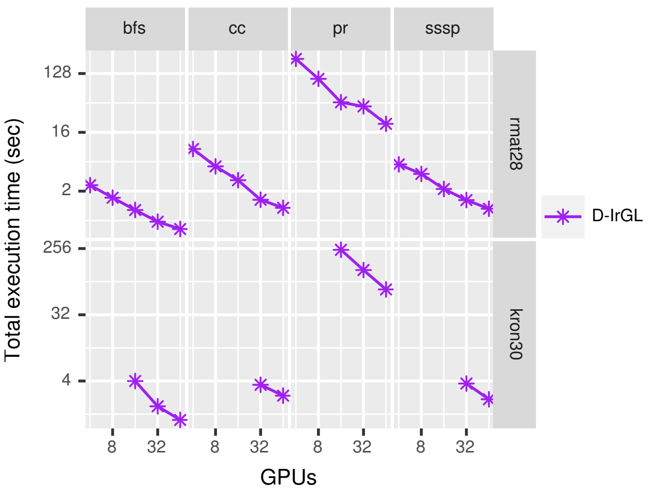


**Figure 9.** Strong scaling (log-log scale) of D-IrGL on the Bridges supercomputer (4 K80 GPUs share a physical node).

bfs on clueweb12. Both D-Ligra and D-Galois send an order of magnitude less data than Gemini due to the communication optimizations in Gluon and the more flexible partitioning strategies supported by this system. The only exception is pr on clueweb12. In this case, D-Galois outperforms Gemini because CVC sends fewer messages and has fewer communication partners than the edge-cut on Gemini even though the communication volume is greater. Thus, D-Galois is an order of magnitude faster than Gemini on 128 or more hosts.

To analyze load imbalance, we measure the computation time of each round on each host, calculate the maximum and mean across hosts of each round, and sum them over rounds to determine the maximum computation time and mean computation time, respectively. The value of maximum-by-mean computation time yields an estimate of the overhead due to load imbalance: the higher the value, the more the load imbalance. On 128 and 256 hosts for cc and pr in D-Galois with clueweb12 and wdc12, the overhead value ranges from ~3 to ~8, indicating a heavily imbalanced load. The overhead value in D-Ligra for the same cases ranges from ~3 to ~13, indicating that it is much more imbalanced; D-Ligra is also heavily imbalanced for sssp. In all other cases on 128 and 256 hosts, both D-Galois and D-Ligra are relatively well balanced with the overhead value being less than ~2.5.

## 5.5 Strong Scaling of Distributed GPU System

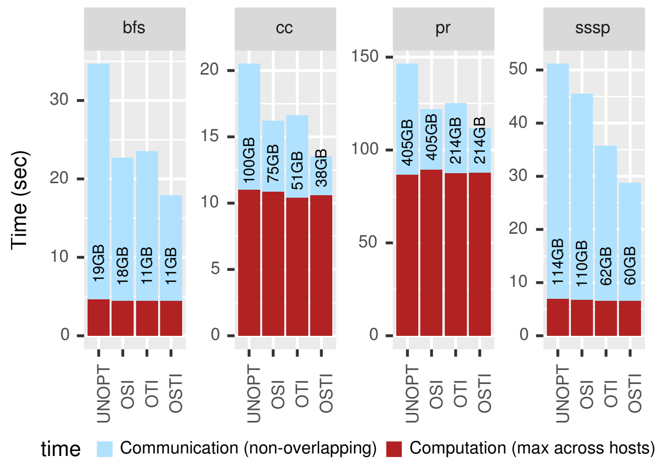
Gluon enables us to build D-IrGL, which is the first multi-node, multi-GPU graph analytics system. We first evaluate its single-node performance by comparing it with Gunrock (March 11, 2018 commit), the state-of-the-art single-node multi-GPU graph analytics system, for rmat26 and twitter40 (Gunrock runs out-of-memory for rmat28 or larger graphs) on a platform with four GPUs sharing a physical node. Like other existing multi-GPU graph analytical systems [8, 74], Gunrock can handle only outgoing edge-cuts[5]. We evaluated D-IrGL with the partitioning policies described in Section 5.2. Table 5 shows the results. Although Gunrock performs better than D-IrGL using OEC in most cases, D-IrGL outperforms Gunrock by using more flexible partitioning policies. Considering the best-performing partitioning policy, D-IrGL gives a geomean speedup of 1.6× over Gunrock.

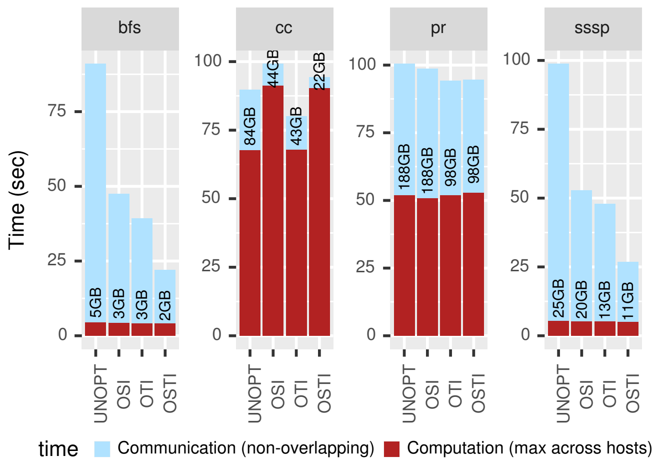
Figure 9 shows the strong scaling of D-IrGL[6]. For rmat28, D-IrGL on 64 GPUs yields a geomean speedup of ~6.5× over that on 4 GPUs. The key takeaway is that Gluon permits D-IrGL to scale well like Gluon-based CPU systems.

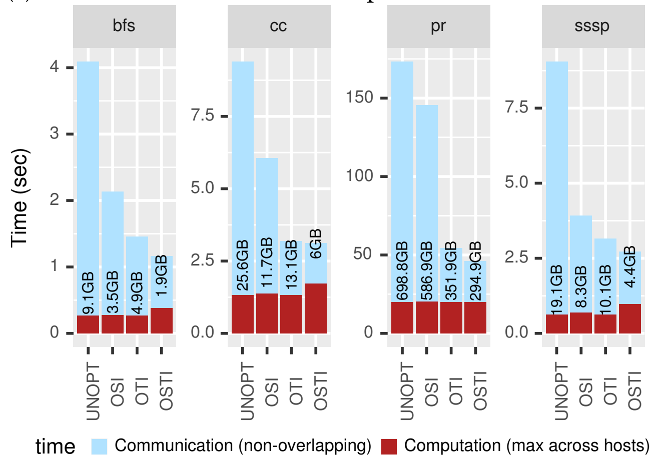
## 5.6 Analysis of Gluon's Communication Optimizations

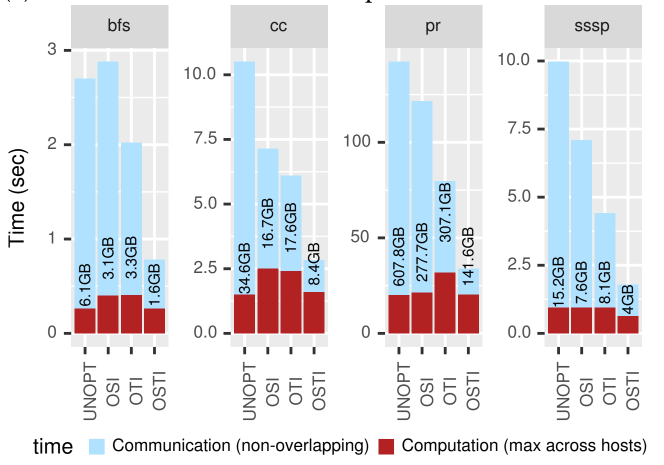
To understand the impact of Gluon's communication optimizations, we present a more detailed analysis of D-Galois on 128 KNL nodes of Stampede for clueweb12 using CVC and OEC, D-IrGL on 16 GPUs of Bridges for rmat28 using CVC and IEC, and D-IrGL on 4 GPUs of Bridges for twitter40 using CVC and IEC. Figure 10 shows their execution time with several levels of communication optimization. In UNOPT, optimizations that exploit structural invariants (Section 3) and temporal invariance (Section 4) are disabled. Optimizations exploiting structural invariants and optimizations exploiting temporal invariance are enabled in OSI and OTI, respectively, while OSTI is the standard Gluon system with both turned on[7]. Each bar shows the breakdown into computation time and communication time, and within each bar, we show the communication volume. We measured the computation time of each round on each host, take the maximum across hosts in each round, and sum them over rounds, which is reported as the (maximum) computation time in Figure 10. The rest of the execution time is reported as the (non-overlapping) communication time. As a consequence, the load imbalance would be incorporated in the computation time.

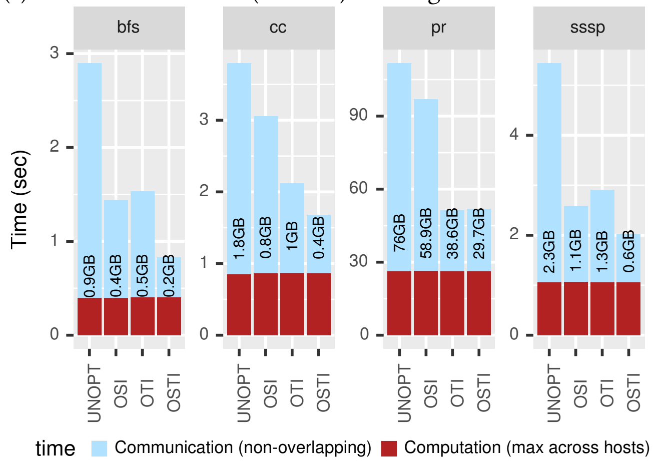
[5]We evaluated the different OEC policies in Gunrock and chose the best performing one for each benchmark and input (mostly, random edge-cut).
[6]rmat26 and twitter40 are too small while clueweb12 and wdc12 are too large to fit on fewer GPUs. Missing point indicates out-of-memory.
[7]In these D-IrGL experiments, we introduce cudaDeviceSynchronize(), after each computation kernel, to measure computation time precisely, so OSTI results for D-IrGL might differ slightly from that of the standard D-IrGL.


(a) D-Galois on 128 hosts of Stampede: clueweb12 with CVC


(b) D-Galois on 128 hosts of Stampede: clueweb12 with OEC


(c) D-IrGL on 16 GPUs (4 nodes) of Bridges: rmat28 with CVC


(d) D-IrGL on 16 GPUs (4 nodes) of Bridges: rmat28 with IEC


(e) D-IrGL on 4 GPUs (1 node) of Bridges: twitter40 with CVC

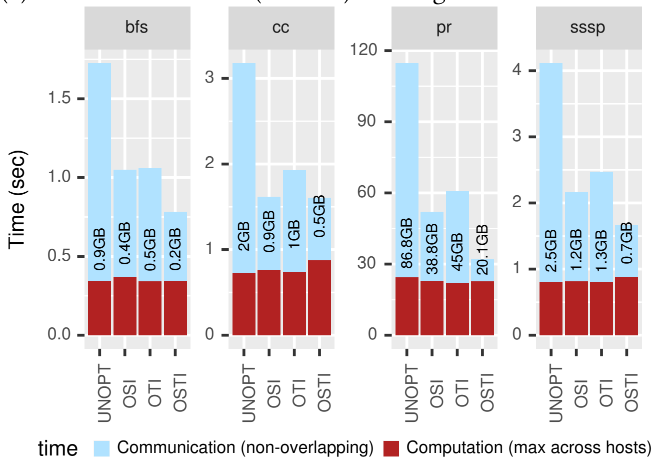

(f) D-IrGL on 4 GPUs (1 node) of Bridges: twitter40 with IEC

**Figure 10.** Gluon's communication optimizations (O): SI - structural invariants, TI - temporal invariance, STI - both SI and TI.

The first high-level observation is that, as expected, communication time is usually a significant portion of the overall execution time in all benchmarks even with all communication optimizations enabled. For cc and pr for clueweb12 on 128 KNL nodes, the computation time seems to be more than the communication time, but this is due to the load imbalance in those applications as explained earlier. The second high-level observation is that OTI has a significant impact on reducing communication volume. UNOPT sends 32-bit global-IDs along with the data, which is 32-bit in all cases. In OTI, memoization permits Gluon to send a bit-vector instead of global-IDs, reducing the communication volume by ~2×. UNOPT also has the time overhead of translating to and from global-IDs during communication; this has more impact on the GPUs since this is done on the host CPU. OSI plays a significant role in reducing communication costs too. On 128 hosts using the CVC policy, UNOPT results in broadcasting updated values to at the most 22 hosts while OPT broadcasts it to at the most 7 hosts only. The overhead of these optimizations (memoization) is small: the mean runtime overhead is ~4% of the execution time, and the mean memory overhead is ~0.5%. Due to these optimizations, OSTI yields a geomean speedup of ~2.6× over UNOPT.

### 5.7 Discussion

Systems like Gemini or Gunrock can be interfaced with Gluon to improve their communication performance. Although Gluon supports heterogeneous devices, we do not report distributed CPUs+GPUs because the 4 GPUs on a node on Bridges outperform the CPU by a substantial margin. Nonetheless, Gluon enables plugging-in IrGL and Ligra or Galois to build distributed, heterogeneous graph analytics systems in which the device-optimized computation engine can be chosen at runtime.

## 6 Related Work

***Shared-memory CPU and single-GPU systems.*** Galois [25, 53, 58], Ligra [62], and Polymer [73] are state-of-the-art graph analytics frameworks for multi-core NUMA machines which have been shown to perform much better than existing distributed graph analytics systems when the graph fits in the memory of a node [44]. Gluon is designed to scale out these efficient shared-memory systems to distributed-memory clusters. As shown in Table 3, Gluon scales out Ligra (D-Ligra) and Galois (D-Galois) to 256 hosts. Single-GPU frameworks [24, 26, 28, 34, 55, 69] and algorithm implementations [14, 45, 49–51] have shown that the GPU can be efficiently utilized for irregular computations.

***Single-node multi-GPUs and heterogeneous systems.*** Several frameworks or libraries exist for graph processing on multiple GPUs [8, 21, 46, 56, 74] and multiple GPUs with a CPU [18, 22, 41]. Groute [8] is asynchronous; the rest of them use BSP-style synchronization. The most important limitation of these systems is that they are restricted to a single machine, so they cannot be used for clusters in which each machine is a multi-GPU system. This limits the size of graphs that can be run on these systems. In addition, they only support outgoing edge-cut partitions. D-IrGL is the first system for graph analytics on clusters of multi-GPUs.

***Distributed in-memory systems.*** Many systems [2, 13, 17, 23, 27, 29, 30, 33, 38, 43, 52, 68, 70–72, 75] exist for distributed CPU-only graph analytics. All these systems are based on variants of the vertex programming model. Gemini [75] is the state-of-the-art, but it does not scale well since it does not optimize the communication volume like Gluon is able to, as seen in Figure 8. Moreover, computation is tightly coupled with communication in most of these systems, precluding them from using existing efficient shared-memory systems as their computation engine. Some of them like Gemini and LFGraph [29] only support edge-cut partitioning policies, but as we see in our evaluation, vertex-cut partitioning policies might be needed to scale well. Although the others handle unconstrained vertex-cuts, they do not optimize communication using structural or temporal invariants in the partitioning. Gluon's communication optimizations can be used in all these systems to build faster systems.

***Out-of-core systems.*** Many systems [35, 37, 42, 60, 61, 76] exist for processing graphs directly from secondary storage. GTS [35] is the only one which executes on GPUs. Chaos [60] is the only one which runs on a distributed cluster. Although they process graphs that do not fit in memory, their solution is orthogonal to ours.

***Graph partitioning.*** Gluon makes the design space of existing partitioning policies [7, 11, 12, 15, 17, 23, 31, 32, 39, 57, 63, 66] easily available to the graph applications. Cartesian vertex-cut is a novel class of partitioning policies we identified that can reduce the communication volume and time over more general vertex-cut partitioning policies.

## 7 Conclusions

This paper described Gluon, a communication-optimizing substrate for distributed graph analytics that supports heterogeneity in programming models, partitioning policies, and processor types. Gluon can be used to scale out existing shared-memory CPU or GPU graph analytical systems. We show that such systems outperform the state-of-the-art. The communication optimizations in Gluon can also be used in existing distributed-memory systems to make them faster. The source code for Gluon, D-Galois, and D-IrGL are publicly available along with Galois v4.0 [3].

## Acknowledgments

This research was supported by NSF grants 1337217, 1337281, 1406355, 1618425, 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Bridges system, supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center, and the Stampede system at Texas Advanced Computing Center, University of Texas at Austin.

## References

[1] 2010. Graph 500 Benchmarks. http://www.graph500.org

[2] 2013. Apache Giraph. http://giraph.apache.org/

[3] 2018. The Galois System. http://iss.ices.utexas.edu/?p=projects/galois

[4] 2018. The Lonestar Benchmark Suite. http://iss.ices.utexas.edu/?p=projects/galois/lonestar

[5] 2018. Pittsburgh Supercomputing Center (PSC). https://www.psc.edu/

[6] 2018. Texas Advanced Computing Center (TACC), The University of Texas at Austin. https://www.tacc.utexas.edu/

[7] Amine Abou-Rjeili and George Karypis. 2006. Multilevel Algorithms for Partitioning Power-law Graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, USA, 124–124. http://dl.acm.org/citation.cfm?id=1898953.1899055

[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/3018743.3018756

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[10] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[11] E. G. Boman, K. D. Devine, and S. Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503293

[12] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced Graph Edge Partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 1456–1465. https://doi.org/10.1145/2623330.2623660

[13] Aydin Buluc and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[14] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. 141–151.

[15] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. 2010. On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe. *SIAM J. Sci. Comput.* 32, 2 (Feb. 2010), 656–683. https://doi.org/10.1137/080737770

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. *R-MAT: A Recursive Model for Graph Mining*. 442–446. https://doi.org/10.1137/1.9781611972740.43

[17] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[18] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. 2016. Falcon: A Graph Manipulation Language for Heterogeneous Systems. *TACO* 12, 4 (2016), 54. https://doi.org/10.1145/2842618

[19] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein (Eds.). 2001. *Introduction to Algorithms.* MIT Press.

[20] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. 2018. A Lightweight Communication Runtime for Distributed Graph Analytics. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[21] Erich Elsen and Vishal Vaidyanathan. 2014. VertexAPI2 – A Vertex-Program API for Large Graph Computations on the GPU. (2014). www.royal-caliber.com/vertexapi2.pdf

[22] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, 345–354.

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

[24] W. Han, D. Mawhirter, B. Wu, and M. Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 233–245. https://doi.org/10.1109/PACT.2017.41

[25] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1941553.1941557

[26] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan. 2017. Multi-Graph: Efficient Graph Processing on GPUs. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 27–40. https://doi.org/10.1109/PACT.2017.48

[27] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 58, 12 pages. https://doi.org/10.1145/2807591.2807620

[28] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 267–276. https://doi.org/10.1145/1941553.1941590

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

[30] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. Graph-Builder: Scalable Graph ETL Framework. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2484425.2484429

[31] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. https://doi.org/10.1137/S1064827595287997

[32] George Karypis and Vipin Kumar. 1999. Multilevel K-way Hypergraph Partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99)*. ACM, New York, NY, USA, 343–348. https://doi.org/10.1145/309847.309954

[33] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th*

*ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 169–182. https://doi.org/10.1145/2465351.2465369

[34] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[35] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A Fast and Scalable Graph Processing Method Based on Streaming Topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 447–461.

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

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[38] Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 278–289. https://doi.org/10.1109/ICDE.2013.6544832

[39] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. 2015. Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 56, 12 pages. https://doi.org/10.1145/2807591.2807632

[40] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* 11 (March 2010), 985–1042. http://dl.acm.org/citation.cfm?id=1756006.1756039

[41] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 195–207. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

[42] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 527–543. https://doi.org/10.1145/3064176.3064191

[43] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Intl Conf. on Management of Data (SIGMOD '10)*. 135–146. https://doi.org/10.1145/1807167.1807184

[44] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=2831090.2831104

[45] Mario Mendez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-based Points-to Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. http://iss.ices.utexas.edu/Publications/Papers/oopsla10-mendezlojo.pdf

[46] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/2145816.2145832

[47] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2012. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/

[48] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM, New York, NY, USA, 427–432. https://doi.org/10.1145/2567948.2576928

[49] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/2458523.2458533

[50] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS '13)*. Springer-Verlag, London, UK.

[51] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA.

[52] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 291–305. http://dl.acm.org/citation.cfm?id=2813767.2813789

[53] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[54] Nicholas A. Nystrom, Michael J. Levine, Ralph Z. Roskies, and J. Ray Scott. 2015. Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure (XSEDE '15)*. ACM, New York, NY, USA, Article 30, 8 pages. https://doi.org/10.1145/2792745.2792775

[55] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015

[56] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU Graph Analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 479–490. https://doi.org/10.1109/IPDPS.2017.117

[57] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. ACM, New York, NY, USA, 243–252. https://doi.org/10.1145/2806416.2806424

[58] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '11)*. 12–25. https://doi.org/10.1145/1993498.1993501

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

[60] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 410–424. https://doi.org/10.1145/2815400.2815408

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[62] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '13)*. 135–146. https://doi.org/10.1145/2442516.2442530

[63] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. ACM, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722

[64] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. ACM, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/3093338.3093385

[65] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. 2014. XSEDE: accelerating scientific discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74.

[66] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM '14)*. ACM, New York, NY, USA, 333–342. https://doi.org/10.1145/2556195.2556213

[67] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[68] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 861–878. https://doi.org/10.1145/2660193.2660227

[69] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

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

[71] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux$^2$: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 669–682. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/xiao

[72] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*.

[73] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 183–193. https://doi.org/10.1145/2688500.2688507

[74] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014). https://doi.org/10.1109/TPDS.2013.111

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

[76] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu