Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics

Roshan Dathathri*, Gurbinder Gill†, Loc Hoang§, V. Krishna Nandivada
Vishwesh Jatala‡, Keshav Pingali**
University of Texas at Austin
{*roshan, †gill, §loc, **pingali}@cs.utexas.edu,
‡vishwesh.jatala@austin.utexas.edu

IIT Madras
nvk@iitm.ac.in

Hoang-Vu Dang, Marc Snir
University of Illinois at Urbana-Champaign
{hdang8, snir}@illinois.edu

***Abstract*—Distributed graph analytics systems for CPUs, like D-Galois and Gemini, and for GPUs, like D-IrGL and Lux, use a bulk-synchronous parallel (BSP) programming and execution model. BSP permits bulk-communication and uses large messages which are supported efficiently by current message transport layers, but bulk-synchronization can exacerbate the performance impact of load imbalance because a round cannot be completed until every host has completed that round. Asynchronous distributed graph analytics systems circumvent this problem by permitting hosts to make progress at their own pace, but existing systems either use global locks and send small messages or send large messages but do not support general partitioning policies such as vertex-cuts. Consequently, they perform substantially worse than bulk-synchronous systems. Moreover, none of their programming or execution models can be easily adapted for heterogeneous devices like GPUs.**

In this paper, we design and implement a lock-free, non-blocking, bulk-asynchronous runtime called Gluon-Async for distributed and heterogeneous graph analytics. The runtime supports any partitioning policy and uses bulk-communication. We present the bulk-asynchronous parallel (BASP) model which allows the programmer to utilize the runtime by specifying only the abstract communication required. Applications written in this model are compared with the BSP programs written using (1) D-Galois and D-IrGL, the state-of-the-art distributed graph analytics systems (which are bulk-synchronous) for CPUs and GPUs, respectively, and (2) Lux, another (bulk-synchronous) distributed GPU graph analytical system. Our evaluation shows that programs written using BASP-style execution are on average $\sim 1.5\times$ faster than those in D-Galois and D-IrGL on real-world large-diameter graphs at scale. They are also on average $\sim 12\times$ faster than Lux. To the best of our knowledge, Gluon-Async is the first asynchronous distributed GPU graph analytics system.

***Index Terms*—Graph analytics, distributed and heterogeneous, BSP model, asynchronous parallel execution models.**

I. Introduction

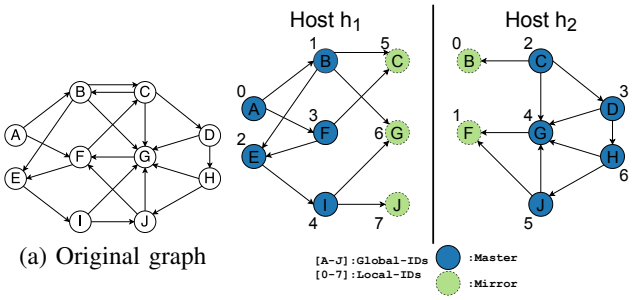
Present-day graph analytics systems have to handle large graphs with billions of nodes and trillions of edges [1]. Since graphs of this size may not fit in the main memory of a single machine, systems like Pregel [2], PowerGraph [3], Gemini [4], D-Galois [5], D-IrGL [5], and Lux [6] use distributed-memory clusters. In these distributed graph analytics systems, the graph is partitioned [7], [8], [9] so that each partition fits in the memory of one host in the cluster, and the *bulk-synchronous parallel* (BSP) programming model [10] is used. In this model, the program is executed in rounds, and each round consists of computation followed by communication. In the computation phase, each host updates node labels in its partition. In the communication phase, boundary node labels are reconciled so all hosts have a consistent view of labels. The algorithm terminates when a round is performed in which no label is updated on any host.

One drawback of the BSP model is that it can exacerbate the performance impact of load imbalance because a round cannot be completed until every host has completed that round. This happens frequently in graph analytics applications for two reasons: (1) unstructured power-law graphs are difficult to partition evenly, and (2) efficient graph analytics algorithms are *data-driven* algorithms that may update different subsets of nodes in each round [11], making static load balancing difficult.

One solution is to use *asynchronous* programming models and systems [12], [13], [14], [15], [16], [17], which take advantage of the fact that many graph analytics algorithms are robust to stale reads. Here, the notion of rounds is eliminated, and a host performs computation at its own pace while an underlying messaging system ingests messages from remote hosts and incorporates boundary node label updates into the local partition of the graph. Asynchronous algorithms for particular problems like single-source shortest-path (sssp) [18] and graph coloring [19] have also been implemented. Some of these systems or implementations use global locks or send small messages, but current communication substrates in large clusters are engineered for large message sizes. The other systems send large messages but either do not handle general partitioning policies like vertex-cuts [20], [21] or do not optimize communication [5]. Consequently, the performance of these systems is not competitive with BSP systems like Gemini [4] or D-Galois [5]. In addition, it is not straightforward to extend these asynchronous programming or execution models to execute on heterogeneous devices like GPUs.

In this paper, we explore a novel lock-free, non-blocking, asynchronous programming model that we call *bulk*-asynchronous parallel (BASP), which aims to combine the advantages of bulk communication in BSP models with the computational progress advantages of asynchronous models. BASP retains the notion of a round, but a host is not required to

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00010



(a) Original graph

(b) Partitioned graph

Fig. 1: An example of partitioning a graph (Source: Gluon [5]).

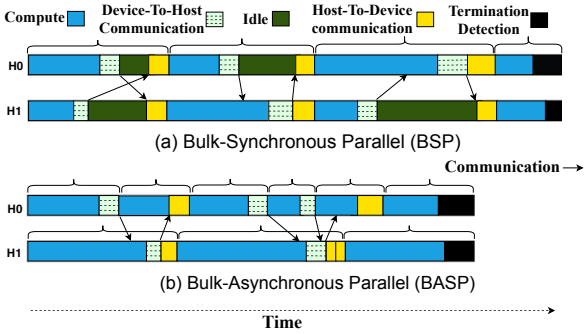


Fig. 2: BSP vs. BASP execution.

wait for other hosts when computation in a round is completed; instead, it sends and receives messages (if available) and moves on to the next round. One advantage of the BASP model is that it is relatively easy to modify BSP programs to BASP programs. It is also easy to modify BSP-based graph analytics systems for CPUs or GPUs to implement this model.

In our study, we use D-Galois and D-IrGL [5], the state-of-the-art distributed CPU and GPU graph analytics systems, respectively. Both these systems are built using the communication-optimizing substrate, Gluon [5]. By modifying Gluon to support the BASP model, we develop the first asynchronous, distributed, heterogeneous graph analytics system; we name this system Gluon-Async. Like Gluon, Gluon-Async can be used to extend or compile [22] existing shared-memory CPU-only or GPU-only graph analytical systems for distributed and heterogeneous execution. For large-diameter real-world web-crawls, Gluon-Async is on an average $\sim 1.4\times$ faster than D-IrGL on 64 GPUs and $\sim 1.6\times$ faster than D-Galois on 128 hosts. Furthermore, it is $\sim 12\times$ faster than Lux, another BSP-style distributed GPU graph analytics system.

The rest of this paper is organized as follows. Section II gives an overview of BSP-style distributed graph analytics and introduces the BASP model. Section III shows how Gluon [5], the state-of-the-art BSP-style distributed and heterogeneous graph analytics system, can be converted to BASP-style execution, and we believe similar modifications can be made to other BSP-based systems. Section IV gives experimental results on Stampede2, a large CPU cluster, and on Bridges, a distributed multi-GPU cluster. Section V describes the related work, and Section VI summarizes the results of this study.

II. Bulk-Asynchronous Parallel Model

This section introduces the BASP model. We start with an overview of the BSP model before describing BASP.

A. Overview of Bulk-Synchronous Parallel (BSP) Execution

At the start of the computation, the graph is partitioned among the hosts using one of many partitioning policies [20]. Figure 1 shows a graph that has been partitioned between two hosts. The edges of the graph are partitioned between hosts, and proxy nodes are created on each host for the end-points of its edges. Since the edges connected to a given vertex may be mapped to different hosts, a given vertex in the graph may have proxies on several hosts. One of these proxies is designated the master, and the others are designated as mirrors. During computation, the master holds the canonical value of the vertex, and it communicates that value to the mirrors when needed. In Figure 1, host h_1 has masters for nodes $\{$A,B,E,F,I$\}$ and mirrors for nodes $\{$C,G,J$\}$.

Execution of the program occurs in rounds. In each round, a host computes independently on its partition of the graph. Most existing systems use the *vertex programming model* in which nodes either update the labels of their neighbors (push-style operator) or update their own labels using the labels of their neighbors (pull-style operator) until quiescence is reached. Since a vertex in the original graph can have proxies on several hosts, the labels of these proxies may be updated differently on different hosts. For example, in a push-style breadth-first search (BFS) computation on the graph of Figure 1 rooted at vertex A, the mirror vertex for G on host h_1 may get the label 2 from B while the master vertex for G on host h_2 remains at the initial value ∞.

To reconcile these differences, it is necessary to perform inter-host communication. A key property of many graph analytics algorithms is that the differences among the labels of vertices can be reconciled by communicating the labels of all mirrors to the master, reducing them using an application-dependent operation, and broadcasting the result to all mirrors (as each edge is present on only one host, updates to edge labels do not involve communication). In the BFS example considered above, the value 2 will be sent to the master for vertex G on host h_2 where it is reduced with the master's label using the "minimum" operation, and the result 2 is used to update the labels of the master and mirrors. This pattern of reconciling labels using a reduction operation at the master followed by broadcast to mirrors can be used for any partitioning strategy [5]. It can also be used to offload the computation on any device [5].

In the BSP model, this reconciliation of node labels by inter-host communication is performed in each round of execution, and a host must send and ingest *all* updates from other hosts in that round before it can proceed to the next round. As a consequence, the slowest, or *straggler*, host in a round deter-

```
Graph* g;
struct GNode { // data on each node
  uint32_t dist_old;
  uint32_t dist_cur;
};
gluon::DistAccumulator<unsigned int> terminator;
...// sync structures
struct SSSP {
  void operator()(GNode src) const {
    if (src.dist_old > src.dist_cur) {
      terminator += 1; // do not terminate
      src.dist_old = src.dist_cur;
      for (auto dst : g->neighbors(src)) {
        uint32_t new_dist;
        new_dist = src.dist_cur + g->weight(src,dst)
    ;
        atomicMin(dst.dist_cur, new_dist);
      }
    }
  }
};
...// initialization, 1st round for source
do { // filter-based data-driven rounds
  terminator.reset();
  galois::do_all(g->begin(), g->end(), SSSP{&g});
  gluon::sync< .../* sync structures */ >();
} while(terminator.reduce());
```

Fig. 3: Single source shortest path (sssp) application in BSP programming model.

```
Graph* g;
struct GNode { // data on each node
  uint32_t dist_old;
  uint32_t dist_cur;
};
gluon::DistTerminator<unsigned int> terminator;
...// sync structures
struct SSSP {
  void operator()(GNode src) const {
    if (src.dist_old > src.dist_cur) {
      terminator += 1; // do not terminate
      src.dist_old = src.dist_cur;
      for (auto dst : g->neighbors(src)) {
        uint32_t new_dist;
        new_dist = src.dist_cur + g->weight(src,dst)
    ;
        atomicMin(dst.dist_cur, new_dist);
      }
    }
  }
};
...// initialization, 1st round for source
do { // filter-based data-driven rounds
  terminator.reset();
  galois::do_all(g->begin(), g->end(), SSSP{&g});
  gluon::try_sync< .../* sync structures */ >();
} while(terminator.cannot_terminate());
```

Fig. 4: sssp application in BASP programming model. The modifications with respect to Figure 3 are highlighted.

mines when all hosts complete that round. This may increase the *idle* time of the other hosts and lead to load imbalance among hosts. This is exacerbated when the algorithm requires 100s of bulk-synchronous rounds to converge. Large real-world graph datasets have non-trivial diameter which may execute for several rounds in the BSP model. This is turn may result in load imbalance among hosts, hurting performance (we analyze this in Section IV-D). One way to overcome this is to relax the bulk-synchronization required in each round.

B. Overview of Bulk-Asynchronous Parallel (BASP) Execution

The *bulk-asynchronous parallel* (BASP) execution model is based on the following intuition: *when a host completes its computation in a round, it can send messages to other hosts and ingest messages from other hosts, but it can go on to the next round of computation without waiting for messages from any stragglers.* Conceptually, the barrier at the end of each BSP round becomes a point at which each host sends and ingests messages without waiting for all other hosts to reach that point. The correctness of this execution strategy depends on the fact that graph analytics algorithms are resilient to stale reads: as long as there are no lost updates, execution will complete correctly.

Since hosts perform communication only at the end of a round, the BASP execution model permits the message transport layer to use large messages, which is advantageous on current systems since they do not handle small messages efficiently. In contrast, the asynchronous model in GraphLab [12] uses small messages (along with locks) to interleave inter-host communication with computation, which is difficult to support efficiently on current systems.

Figure 2(a) shows a timeline for BSP-style computation on two GPUs. Each GPU is assumed to be a device that is connected to a host that performs inter-host communication. In each round, a GPU performs computation, transfers data to its host, and gets data from its host when that host receives it from the remote host. One feature of efficient graph analytics algorithms is that the amount of computation in each round in a given partition can vary unpredictably between rounds, so balancing computational load statically is difficult. This means that in each BSP round, some GPUs may be idle for long periods of time waiting for overloaded GPUs to catch up. This is shown in the second BSP round in Figure 2(a): device H1 has more computation to do than device H0 in some rounds (and vice-versa), so in those rounds, one host must idle or wait for the other host to finish and send its data. Figure 2(b) illustrates the same computation under the BASP-model: here, the idle time has been completely eliminated.

While BASP exploits the resilience of graph analytics programs to stale reads to compensate for lack of load balance, stale reads may result in wasted computation. For example, under BSP execution, a host may ingest an update from another host and compute immediately with that value in the next round, whereas under BASP execution, the host may miss the update, compute with the stale value, and see the update only in a later round at which point it will need to repeat the computation with the updated value. Therefore, if load is already well-balanced under BSP execution, BASP

execution may not be advantageous. We study these trade-offs by building and analyzing a BASP system.

III. Adapting Bulk-Synchronous Systems for Bulk-Asynchronous Execution

In this section, we describe how we adapted a BSP-style distributed and heterogeneous graph analytics system for BASP execution using the state-of-the-art communication substrate Gluon [5]. We first describe the changes required to Gluon application programs to make them amenable to BASP execution (Section III-A). We then describe changes to Gluon to support BASP-style execution (Section III-B). We use the terms Gluon-Sync and Gluon-Async to denote BSP-style and BASP-style Gluon, respectively. Finally, we present a non-blocking termination detection algorithm that is required for BASP-style execution (Section III-C). Based on our experience, we believe that other BSP systems can also be easily adapted to BASP.

A. Bulk-Asynchronous Programs

D-Galois [5] is the state-of-the-art distributed graph analytical system for CPUs. D-Galois programs are shared-memory Galois [23] programs that make calls to the Gluon(-Sync) communication substrate to synchronize distributed-memory computation. Figure 3 shows a code snippet for single-source-shortest-path (sssp) application. Each host processes its partition of the graph in rounds: computation is followed by communication. The compute phase (shown at Line 24) processes the vertices in the partitioned graph using a *push-style* operator (shown at Line 9) to compute and update the new distance values for their neighbors. The communication phase uses Gluon's communication interface, *i.e.*, the *sync()* method (shown at Line 25). Gluon is responsible for coordinating the communication among all hosts; at the end of this phase, all hosts have a consistent view of node labels. The application terminates when there is a round in which no host updates a node label. This can be detected using Gluon's distributed accumulator to determine the number of updates among all hosts in a round.

Figure 4 shows the same sssp application in the BASP programming model using Gluon-Async. The changes to the application are highlighted. The *try_sync* (*non-blocking*) call is responsible for coordinating the communication of labels among the hosts asynchronously. It ensures that each host eventually receives all the expected messages; in other words, it ensures that the hosts have a consistent view of node labels eventually. However, the challenge for each host then is to detect the termination of an application. This is handled efficiently using the *cannot_terminate()* method. The *cannot_terminate* (*non-blocking*) call is responsible for terminating if and only if no node labels can be updated on any host[1]. It ensures that no host terminates as long as some host has some computation or communication left to be completed. Since *try_sync*() and *cannot_terminate()* methods are non-blocking in nature, a host that performs synchronization can proceed to next round of computation phase without waiting for the communication process to complete. Thus, it may improve the performance.

While we explain these changes using D-Galois, the changes to other Gluon-based systems are similar because the only lines of code that changed are those related to Gluon. For example, in D-IrGL, the state-of-the-art distributed GPU graph analytical system, an IrGL compiler-generated CUDA kernel is called instead of `galois::do_all`, and the sync structures have CUDA kernels instead of CPU code. None of this needs to be changed to make the program amenable to BASP execution.

All programs that can be run asynchronously in existing distributed graph frameworks like PowerSwitch [14] and GRAPE+ [17] can use BASP. In addition, if a program can be run asynchronously in shared-memory, then it can use BASP on distributed-memory. In shared-memory, BSP programs can be made asynchronous if the program is resilient to stale reads and if computation is independent of the BSP round number. The same condition acts as a pre-requisite for changing BSP programs to BASP programs. For example, betweenness centrality [24] uses round number in its computation and requires BSP-style execution for correctness, so it cannot be changed for BASP-style execution. Most other BSP graph programs that have been used in the evaluation of distributed graph processing systems [3], [4], [5], [6], [22] can be changed to BASP-style execution by changing only a few lines of code.

B. Bulk-Asynchronous Communication

Recall from Section II that algorithm execution in both Gluon-Sync and Gluon-Async is done in local rounds where each round performs bulk-computation followed by bulk-communication. The bulk-communication itself involves a reduce phase followed by a broadcast phase. Thus, each round has 3 phases: computation, reduce, and broadcast. The computation phase is identical in Gluon-Sync and Gluon-Async, but the other phases differ.

The reduce and broadcast phases are blocking in Gluon-Sync and non-blocking in Gluon-Async. In Gluon-Sync, hosts exchange messages in each phase (even if the message is empty) and hosts wait to receive these messages; this acts like an implicit barrier. Messages are sent in the reduce or broadcast phase of Gluon-Async only if there are updates to mirror nodes (empty messages are not required due to relaxation of synchronization barriers) and no host waits to receive a message. The action for the received messages in Gluon-Async depend on whether they were sent in the reduce or broadcast phase. As there are two phases and messages could be delivered out-of-order, we distinguish between messages sent in reduce and broadcast phases using tags. We describe this more concretely next.

Let host h_i have the set of mirror proxies P_i for which the set of master proxies P_a are on host h_a. Let U_i be the set of mirror proxies on h_i that are updated in round r (by

[1]The value set to *DistTerminator* on each host determines whether "no node labels are updated" or another quiescence condition is the termination criteria.

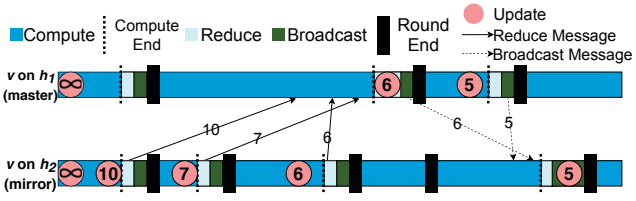


Fig. 5: Illustration of communication in Gluon-Async.

definition, $U_i \subseteq P_i$). Let U_a be the master proxies on h_a that are updated in round r, during either computation or reduce phases (by definition, $U_a \subseteq P_a$).

In Gluon-Sync, the Gluon substrate performs the following operations for every pair of h_i and h_a:

- Reduce phase for h_i: Sends one message m_R containing values of U_i to h_a (if $U_i = \emptyset$, then an empty message is sent) and resets the values of U_i to the identity element of the reduction operation.
- Reduce phase for h_a: *Waits* to receive m_R from h_i and, once received, uses the reduction operator and the values in m_R to update the corresponding master proxies in P_a.
- Broadcast phase for h_a: Sends one message m_B containing values of U_a to h_i (if $U_a = \emptyset$, then an empty message is sent).
- Broadcast phase for h_i: *Waits* to receive m_B from h_a and, once received, uses the values in m_B to set the corresponding mirror proxies in P_i.

To support BASP-style execution of Gluon-Async, we modified the Gluon communication-optimizing substrate to perform the following operations (instead of the above) for every pair of h_i and h_a:

- Reduce phase for h_i: If $U_i \neq \emptyset$, sends a reduce-tagged message m_R containing values of U_i to h_a and resets the values of U_i to the identity element of the reduction operation.
- Reduce phase for h_a: For every reduced-tagged message m_R received from h_i, uses the reduction operator and the values in m_R to update the corresponding master proxies in P_a.
- Broadcast phase for h_a: If $U_a \neq \emptyset$, sends a broadcast-tagged message m_B containing values of U_a to h_i.
- Broadcast phase for h_i: For every broadcast-tagged message m_B received from h_a, uses *the reduction operator* and the values in m_B to update the corresponding mirror proxies in P_i.

If the reduction operator is not used in the broadcast phase of Gluon-Async, algorithms may not yield correct results (or even converge). To illustrate this with an example, we show the synchronization of proxies in Figure 5 for the single-source shortest path (sssp) code in Gluon-Async (shown in Figure 4). The label `dist_current` (shortened as d_c), is reduced during computation using the "minimum" operation. Consider a vertex v with proxies on hosts h_1 and h_2, where the master proxy is on h_1 and the mirror proxy is on h_2. The label d_c is initialized to ∞ on both proxies. Say host h_2 sends values 10, 7, and 6 after executing its local rounds 1, 2, and 3, respectively. Say host h_1 receives all these values in the order 10, 6, and 7 at the end of its round 2. Host h_1, which still has ∞ value for its proxy, reduces the received values one-by-one, yielding the update 6, and broadcasts this value to h_2. Host h_1 reduces its proxy value during computation to 5 and broadcasts it to h_2 after its round 3. Host h_2 receives both these values in the order of 5 and 6. The mirror proxy value on h_2 is 6 (because reset is a *no-op* for minimum operation). If host h_2 had set the received values (in order) like in Gluon-Sync, then the final value of h_2 would be 6, which would be incorrect. Host h_2 instead reduces the received values one-by-one yielding the update 5. The proxies on both hosts are not updated thereafter and thus, both proxies have the same values.

An important point to note is that if the message is not empty, then Gluon-Sync and Gluon-Async send the same message. Gluon-Async thus retains the underlying advantages of Gluon-Sync. Gluon-Async supports any partitioning policy and performs bulk-communication, thereby utilizing Gluon's communication optimizations that exploit structural and temporal invariants in partitioning policies [5]. Gluon-Async can be plugged into different CPU or GPU graph analytics systems to build distributed-memory versions of those systems that use BASP-style execution. As shown in Figure 2, communication between a GPU device and its host is a local operation. Gluon-Async treats this as a blocking operation like Gluon-Sync. While this can be made non-blocking too, it is outside the scope of this paper.

We showed that BASP-style execution can be used in Gluon-Async without any blocking or waiting operations among hosts. The messages, if any, will be eventually delivered. The key to this is that hosts must not terminate until there are messages left to be delivered. This requires non-blocking termination detection, which we explain next.

C. Non-blocking Termination Detection

BASP-style execution requires a more complicated termination algorithm than BSP-style execution. We describe a non-blocking termination detection algorithm that uses snapshots to implement a distributed consensus protocol [25] that does not rely on message delivery order.

The algorithm is based on a state machine maintained on each host. At any point of time, a host is in one of five states: Active (A), Idle (I), Ready-to-Terminate$_1$ (RT_1), Ready-to-Terminate$_2$ (RT_2), and Terminate (T). The goal of termination detection is that a host should move to T if and only if every other host will move to T. We describe state transitions and actions for ensuring this.

Hosts coordinate with each other by taking non-blocking snapshots that are numbered. When a host takes a snapshot n, it broadcasts its current state to other hosts (non-blocking). Once a host h takes the snapshot n, it cannot take the next snapshot $n+1$ until h knows that every other host has taken the snapshot n. In other words, before h takes the next snapshot $n+1$, h should not only have completed the broadcast it

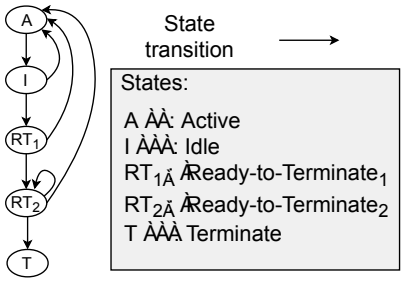


Fig. 6: State transition diagram for termination detection.

Start State	Condition for state transition (boolean formula)	End State	Action
A	inactive	I	
I	¬inactive	A	
I	inactive ∧ inspected	RT_1	Snapshot
RT_1	¬inactive	A	
RT_1	inactive ∧ inspected	RT_2	Snapshot
RT_2	¬inactive	A	
RT_2	inactive ∧ inspected ∧ ¬affirmed	RT_2	Snapshot
RT_2	affirmed	T	Terminate

TABLE I: Conditions required for state transitions during termination detection.

initiated for n but also have received broadcast messages from every other host for n. Thus, eventually, every host will know the states that all other hosts took their snapshots from. For example, all hosts will know whether all hosts took the snapshot n from the same state RT_2 or not. We use this knowledge to transition between states.

Each host has a dedicated communication thread that is started when the program begins (and terminated when program ends). It receives messages throughout program execution. Every host takes a (dummy) snapshot initially. Subsequent snapshots are taken by a host h only if h is ready to terminate. Intuitively, hosts can terminate only if every host knows that "every host knows that every host wants to terminate". This requires two consecutive snapshots to be taken with all hosts indicating that they are ready-to-terminate (RT). We use RT1 and RT2 to distinguish between two consecutive snapshots of RT.

On each host h, the termination detection algorithm is invoked at the end of each local round r; all the state transitions occur only at this point in the program. Note that r is incremented each time *cannot_terminate()* is invoked (see Figure 4 for example). Let n be the last snapshot that h has taken. When the termination detection algorithm is invoked, we first check if h is *inactive*, *inspected*, or *affirmed*.

A host h is considered to be *inactive* if the following conditions hold:

1) No label was updated in round r in computation, reduce, or broadcast phases.
2) All non-blocking sends initiated on this host are complete.
3) All non-blocking receives initiated on this host are complete.

The first condition checks whether work was done in r while the other conditions check whether any work is still pending. These conditions must hold for h to take the next snapshot $n+1$.

A host h is considered to be *inspected* if it knows that all the hosts have taken the previous snapshot n. This condition must hold for h to take the next snapshot $n+1$. Similarly, a host h is considered to be *affirmed* if (i) h has been *inspected* and (ii) it knows that all the hosts have taken the previous snapshot n from state RT_2 (that is, other hosts have also affirmed their readiness to terminate). This condition must hold for h to terminate.

Initially, every host is in state A. Figure 6 shows the possible state transition on a single host. Table I shows the conditions that must hold for each state transition and the action, if any, taken after the state transition. No action is taken with transitions to states A and I. When h transitions to RT_1 or RT_2, it takes a snapshot. When h transitions to T, h decides to terminate (returns false in Line 26 in Figure 4). A host moves from A to I only if the host is *inactive*. If a host is not *inactive*, then it moves to A from the I, RT_1, or RT_2 states. If h is *inspected* and is in I, then it moves to RT_1. If h is *inspected* and is in RT_1, it moves to RT_2. If h is *affirmed*, then it moves from RT_2 to T.

Consider an example with two hosts, h_1 and h_2. Initially, both of them initiate (dummy) snapshot n_0. When h_2 becomes *inactive*, it moves to I. As both hosts initiated the previous snapshot n_0, h_2 moves to RT_1 and initiates the next snapshot n_1. Meanwhile, h_1 sends a message to h_2, becomes inactive, and moves to I. As n_0 has ended, h_1 moves to RT_1 and initiates n_1. In the next round, h_1 detects that h_2 also has initiated n_1. Note that it would be incorrect for h_1 to terminate at this point, although both h_1 and h_2 initiated n_1 from RT_1. Our algorithm uses two RT states to detect this, so h_1 moves to RT_2 instead of terminating and initiates the next snapshot n_2. During this time, h_2 received the message from h_1 which made it *active* and moved it to A. Later, it moves to I and then RT_1 to initiate n_2. In the next round, h_2 observes that n_2 has ended, so it moves to RT_2 and initiates n_3. h_1 also observes that n_2 has ended and initiates n_3 while remaining in RT_2. Now, in the next round on both hosts, each host observes that n_3 has ended and that the other host has initiated n_3 from RT_2, so both hosts *affirm* to terminate and move to T.

To implement our termination detection algorithm in Gluon-Async (Line 26 in Figure 4), we use non-blocking collectives to take a snapshot. For the reduce and broadcast phases, we modify the communication substrate to send messages in *synchronous* mode instead of *standard* mode. In *standard* communication mode of MPI or LCI [26], a send (call) may complete before a matching receive is invoked. Hence, both the sender and the receiver may become inactive and terminate while the message is still in-flight. In contrast, in *synchronous* mode, a send is considered complete only if the receiver has initiated receive. Consequently, when a message is in-flight, either the sender or the receiver is in active state A. Thus, *synchronous* communication mode sends are necessary for our

TABLE II: Input graphs and their key properties (we classify graphs with estimated diameter > 200 as high-diameter graphs).

	Small graphs				Large graphs				
	twitter50	**rmat27**	**friendster**	**uk07**	**gsh15**	**clueweb12**	**uk14**	**wdc14**	**wdc12**
$\|V\|$	51M	134M	66M	106M	988M	978M	788M	1,725M	3,563M
$\|E\|$	1,963M	2,147M	1,806M	3,739M	33,877M	42,574M	47,615M	64,423M	128,736M
$\|E\|/\|V\|$	38	16	28	35	34.3	43.5	60.4	37	36
Max OutDegree	779,958	453M	5,214	15,402	32,114	7,447	16,365	32,848	55,931
Max InDegree	3.5M	21,806	5,214	975,418	59M	75M	8.6M	46M	95M
Estimated Diameter	12	3	21	115	95	498	2,498	789	5,274
Size (GB)	16	18	28	29	260	325	361	493	986

termination detection protocol. Note that our protocol does not rely on the order of message delivery of Gluon or the underlying communication substrate such as MPI or LCI [26].

Note that goal of termination detection is that a host should move to T if and only if every other host will move to T. We now argue how our termination detection algorithm satisfies this property. A non-active, non-terminated host h can move back to state A only if it receives data from another host – in this case, the *inactive* flag will become false. Since the program is correct, at least one host will not reach the RT_2 state until the final value(s) are computed (no false detection of termination). A host h can reach the state RT_2 from RT_1 or RT_2 only if it is *inspected* and *inactive*, which means that h did not update any labels and did not send nor receive data. If every host took the snapshot from RT_2, then no host computed, sent, or received data between two snapshots. Consequently, no host can receive a message and move to A after that, so every host must terminate.

IV. Experimental Evaluation

In this section, we evaluate the benefits of Bulk-Asynchronous Parallel (BASP) execution over Bulk-Synchronous Parallel (BSP) execution using D-Galois [5] and D-IrGL [5], the state-of-the-art graph analytics systems for distributed CPUs and distributed GPUs, respectively. Both these systems are built on top of a Gluon [5]. In this paper, we use the name Gluon-Sync to refer to these two systems. We modified D-Galois and D-IrGL BSP programs as described in Section III-A to make them amenable for BASP-style execution. As described in Sections III-B and III-C, we modified Gluon to support BASP-style execution for both systems, which we call Gluon-Async (source code is publicly available [27]).

We also compare the performance of Gluon-Async with that of Lux [6], which is a multi-host multi-GPU graph analytical framework that uses BSP-style execution; note that there are no asynchronous distributed GPU graph analytical systems to compare against. GRAPE+ [17] and PowerSwitch [14] are asynchronous distributed CPU-only graph systems, and we compare them with Gluon-Async.

We first describe our experimental setup (Section IV-A). We then present our evaluation on distributed GPUs (Section IV-B) and distributed CPUs (Section IV-C). Finally, we analyze BASP and BSP (Section IV-D) and summarize our results (Section IV-E).

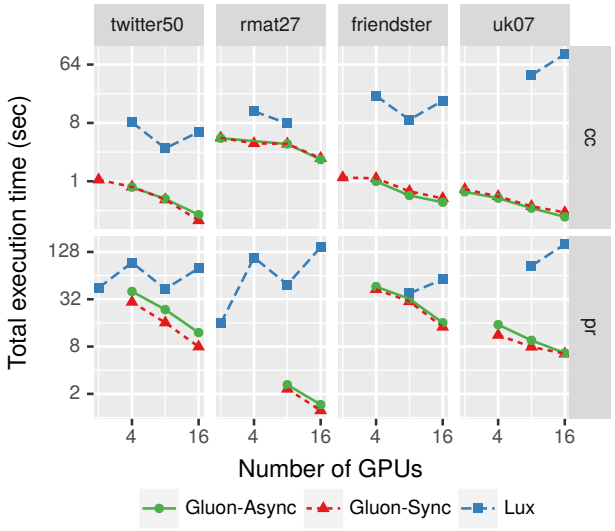


Fig. 7: Strong scaling (log-log scale) of Lux, Gluon-Sync, and Gluon-Async for small graphs on Bridges (2 P100 GPUs share a physical machine).

A. Experimental Setup

We conducted all the GPU experiments on the Bridges cluster [28] at the Pittsburgh Supercomputing Center [29], [30]. Each machine in the cluster is configured with 2 NVIDIA Tesla P100 GPUs and 2 Intel Broadwell E5-2683 v4 CPUs with 16 cores per CPU, DDR4-2400 128GB RAM, and 40MB LLC. The machines are interconnected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). We use up to 64 GPUs (32 machines). All benchmarks were compiled using CUDA 9.2, GCC 7.3, and MVAPICH2 2.3b.

All the CPU experiments were run on the Stampede2 [31] cluster located at the Texas Advanced Computing Center. Each machine is equipped with 2 Intel Xeon Platinum 8160 "Skylake" CPUs with 24 cores per CPU, DDR4 192GB RAM, and 66MB LLC. The machines in the cluster are interconnected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). We use 48 threads on each machine and up to 128 machines (6144 cores or threads). Benchmarks were compiled with GCC 7.1 and IMPI 17.0.3.

Table II shows the input graphs along with their key properties: twitter50 [32], [33] and friendster [34] are social network graphs; rmat27 is a randomized synthetically gener-

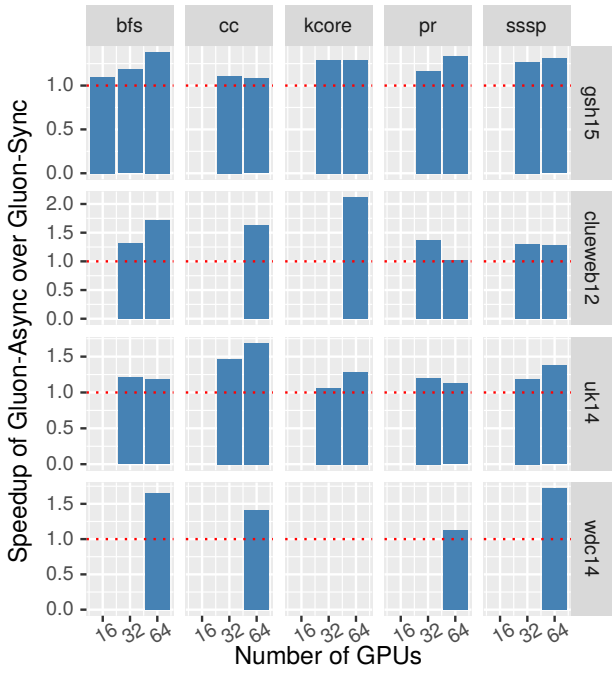


Fig. 8: Speedup of Gluon-Async over Gluon-Sync for large graphs on Bridges (2 P100 GPUs share a physical machine).

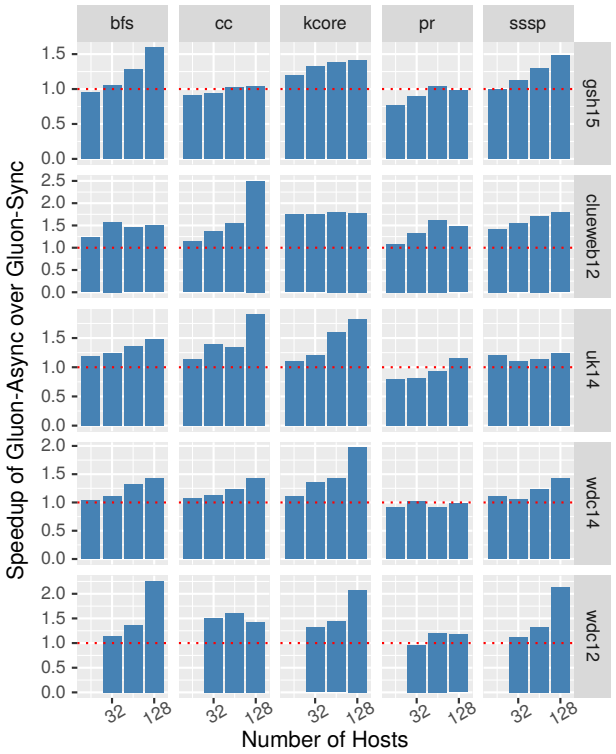


Fig. 9: Speedup of Gluon-Async over Gluon-Sync for large graphs on Stampede (each host is a 48-core Skylake machine).

ated graph using with an RMAT generator [35]; uk07, gsh15, clueweb12 [36], uk14 [32], [33], [37], wdc14, and wdc12 [38] are among the largest public web-crawls (wdc12 is the largest publicly available graph). Table II splits the graphs into two categories: small and large. Small graphs are only used for comparison with Lux, GRAPE+, and PowerSwitch (we could not run these systems using the large graphs), while we use large graphs for all other experiments. We also classify the graphs based on their estimated (observed) diameter. All small graphs are low-diameter graphs with diameter < 200, while all large graphs, except gsh15, are high-diameter graphs with diameter > 200.

We evaluated our framework with 5 benchmarks: breadth-first-search (bfs), connected components (cc), k-core (kcore), pagerank (pr), and single source shortest path (sssp). For pr, we used a tolerance of 10^{-6}. For bfs and sssp, we considered the vertex with maximum out-degree as the source. For kcore, we use a k of 100. All benchmarks are executed until convergence. We report the total execution time, excluding the graph loading, partitioning, and construction time. The reported results are a mean over three runs.

For Gluon-Sync and Gluon-Async, the partitioning policy is configurable as it uses the CuSP streaming partitioner [20]. Based on the recommendations of a large-scale study [21], we choose the Cartesian Vertex Cut (CVC) [9], [5] for all our experiments[2]. We use LCI [26] instead of MPI for message transport among hosts[3].

[2]sssp, clueweb12, GPUs uses Outgoing Edge Cut due to memory limits.

[3]Dang et al. [26] show the benefits of LCI over MPI for graph applications.

For Lux, we only present results for cc and pr as the other benchmarks are not available or produce incorrect output. pr in Lux does not have a convergence criterion, so we executed it for the same number of rounds as that of Gluon-Sync[4] (Gluon-Async might execute more rounds to converge). Note that Lux uses an edge-cut partitioning policy and dynamically re-partitions the graph to balance the load.

GRAPE+ [17] is not publicly available. We present results used in their paper (and provided by the authors). They use a total of 196 cores in their study; to compare with them, we use 12 machines of Stampede with 16 threads (196 cores). They use partitions provided by XtraPulp [8]. They present results only for cc, pr, and sssp on friendster. When comparing with them, we use the same partitioning policy, we use the same source nodes for sssp (5506215, 6556728, 1752217, 3391590, 782658), and we use the same tolerance for pr (10^{-3}). For a relative comparison, we also present the corresponding PowerSwitch [14] results from their paper [17]. We do not evaluate PowerSwitch ourselves because it is an order of magnitude slower.

B. Distributed GPUs

Small graphs: Figure 7 shows the total execution time of Gluon-Async, Gluon-Sync, and Lux on small graphs using up to 16 GPUs. Missing points indicate that the system ran out of

[4]Both Gluon-Sync and Lux are BSP-style and use the same algorithm.

TABLE III: Total execution time of Gluon-Sync and Gluon-Async on 192 cores of Stampede; PowerSwitch and GRAPE+ on 192 cores of a different HPC cluster [17].

Benchmark	Input	PowerSwitch	GRAPE+	Gluon-Sync	Gluon-Async
cc		61.1	10.4	1.7	1.7
pr	**friendster**	85.1	26.4	21.3	21.9
sssp		32.5	12.7	5.8	5.5

TABLE IV: Minimum BSP-rounds for Gluon-Sync on CPUs.

Input	Estimated	Minimum Number of Rounds				
	Diameter	bfs	cc	kcore	pr	sssp
gsh15	95	61	11	239	172	62
clueweb12	498	184	25	696	161	200
uk14	2,498	1,825	80	443	161	1,976
wdc14	789	503	196	146	180	507
wdc12	5,274	2,672	401	277	183	3,953

memory (except for Lux with cc on rmat27 using 16 GPUs, which failed due to a crash). The major trend in the figure is that both Gluon-Async and Gluon-Sync always outperform Lux and scale better. It is also clear that Gluon-Async and Gluon-Sync perform quite similarly. In some cases, Gluon-Async is also noticeably slower (pr on twitter50). We do not expect Gluon-Async to perform better than Gluon-Sync for low-diameter graphs like these because most benchmarks in Gluon-Sync execute very few (< 100) rounds for these. We will analyze this later using larger graphs (Section IV-D). Nevertheless, both Gluon-Async and Gluon-Sync are on average $\sim 12\times$ faster than Lux.

Large Graphs: Figure 8 shows the speedup in total execution time of Gluon-Async over Gluon-Sync for large graphs using up to 64 GPUs (Lux runs out of memory for all the large graphs, even on 64 GPUs). Missing points indicate that either Gluon-Sync or Gluon-Async ran out of memory (almost always, if one runs out of memory, the other also does; only in a couple of cases, Gluon-Async runs out of memory but Gluon-Sync does not because Gluon-Async may use more communication buffers). 64 GPUs are insufficient to load wdc12 as input, partition it, and construct it in memory; so both Gluon-Sync and Gluon-Async run out of memory. It is apparent that Gluon-Async always outperforms Gluon-Sync for large graphs. We observe that the speedup depends on both the input graph and the benchmark. Typically, speedup is better for clueweb12 and wdc14 than gsh15. The speedup is also usually lower for pr than for other benchmarks. We also see that in most cases, the speedup of Gluon-Async over Gluon-Sync increases with an increase in the number of GPUs. This indicates that Gluon-Async scales better than Gluon-Sync. For high-diameter graphs on 64 GPUs, Gluon-Async is on average $\sim 1.4\times$ faster than Gluon-Sync.

C. Distributed CPUs

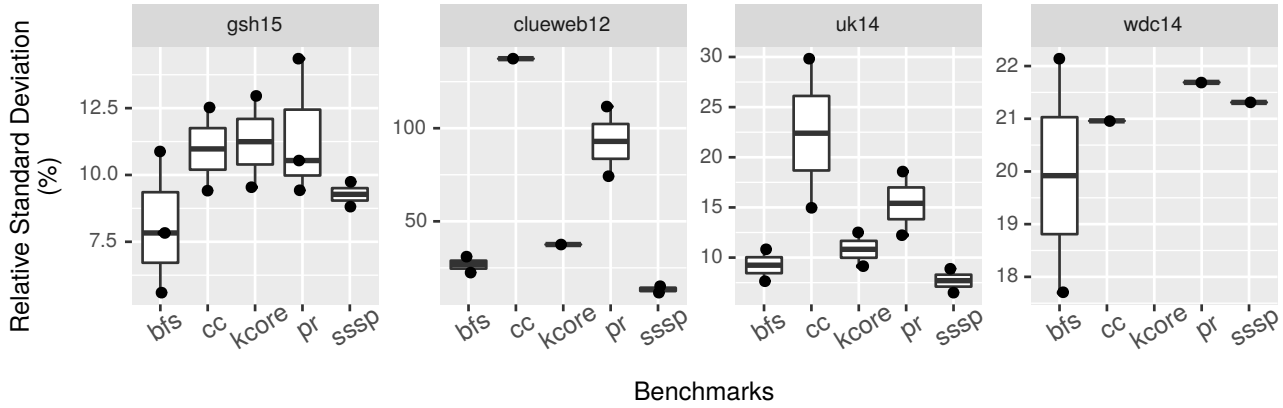
Small graphs: Table III shows the total execution time of PowerSwitch, GRAPE+, Gluon-Sync, and Gluon-Async for friendster with 192 threads. Note that Gluon-Sync and Gluon-Async used machines on Stampede, whereas PowerSwitch and GRAPE+ used machines on a different HPC cluster. Similar to GPUs, the performance differences between Gluon-Async and Gluon-Sync are negligible because friendster is a low-diameter graph. Although both GRAPE+ and PowerSwitch are asynchronous systems, they are much slower than Gluon-Sync and Gluon-Async. Both Gluon-Sync and Gluon-Async are on average $\sim 2.5\times$ and $\sim 9.3\times$ faster than GRAPE+ and PowerSwitch, respectively. This shows that a well-optimized existing bulk-synchronous system (Gluon-Sync) beats the existing asynchronous systems and that it is challenging to reap the benefits of asynchronous execution. Gluon-Sync uses Galois [23] computation engine and Gluon [5] communication engine. Both have several optimizations that help Gluon-Sync outperform PowerSwitch and GRAPE+. It is not straightforward to incorporate these optimizations in PowerSwitch and GRAPE+ due to the way they perform asynchronous communication. Gluon-Async introduces a novel way for asynchronous execution while retaining all the performance benefits of on-device computation engines like Galois and IrGL [39] and the inter-device communication engine, Gluon. While Gluon-Sync and Gluon-Async perform similarly for small graphs, we show that on large graphs, Gluon-Async can be much faster than Gluon-Sync.

Large graphs: Figure 9 shows the speedup in total execution time of Gluon-Async over Gluon-Sync for large graphs using up to 128 Skylake machines or hosts. Missing points indicate that either Gluon-Sync or Gluon-Async ran out of memory. The trends are similar to those on GPUs. The speedup depends on both the input graph and the benchmark. Gluon-Async mostly outperforms Gluon-Sync; its performance is similar or lower than that of Gluon-Sync on 64 or fewer hosts in some cases for pr or in some cases for the input gsh15. The speedup of Gluon-Async over Gluon-Sync increases with the increasing number of hosts indicating that on distributed CPUs also, Gluon-Async scales better than Gluon-Sync. For high-diameter graphs on 128 CPUs, Gluon-Async is on average $\sim 1.6\times$ faster than Gluon-Sync.

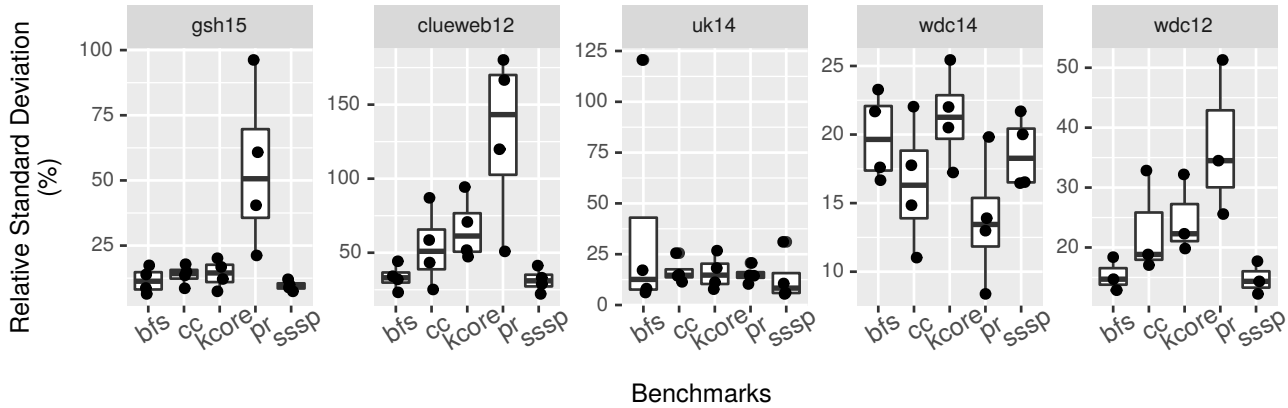
D. Analysis of BASP and BSP

Using Gluon-Async and Gluon-Sync, we now analyze the performance difference between BASP-style and BSP-style execution, respectively, on both distributed GPUs and CPUs. Specifically, we focus on: (1) *why* the difference arises (load imbalance), (2) *where* the difference exists (idle time), and (3) *how* the difference manifests itself (rounds executed).

Load imbalance: Table IV shows the number of rounds executed by benchmarks in Gluon-Sync for the large graphs. It can be observed that higher diameter graphs are likely to execute more rounds, except for pr. We next measure the load imbalance by calculating the total time spent by each host in computation and determine the relative standard deviation (standard deviation by mean) of these values. Figures 10(a) and 10(b) presents these values for Gluon-Sync as a box-



(a) GPUs on Bridges



(a) CPU hosts on Stampede

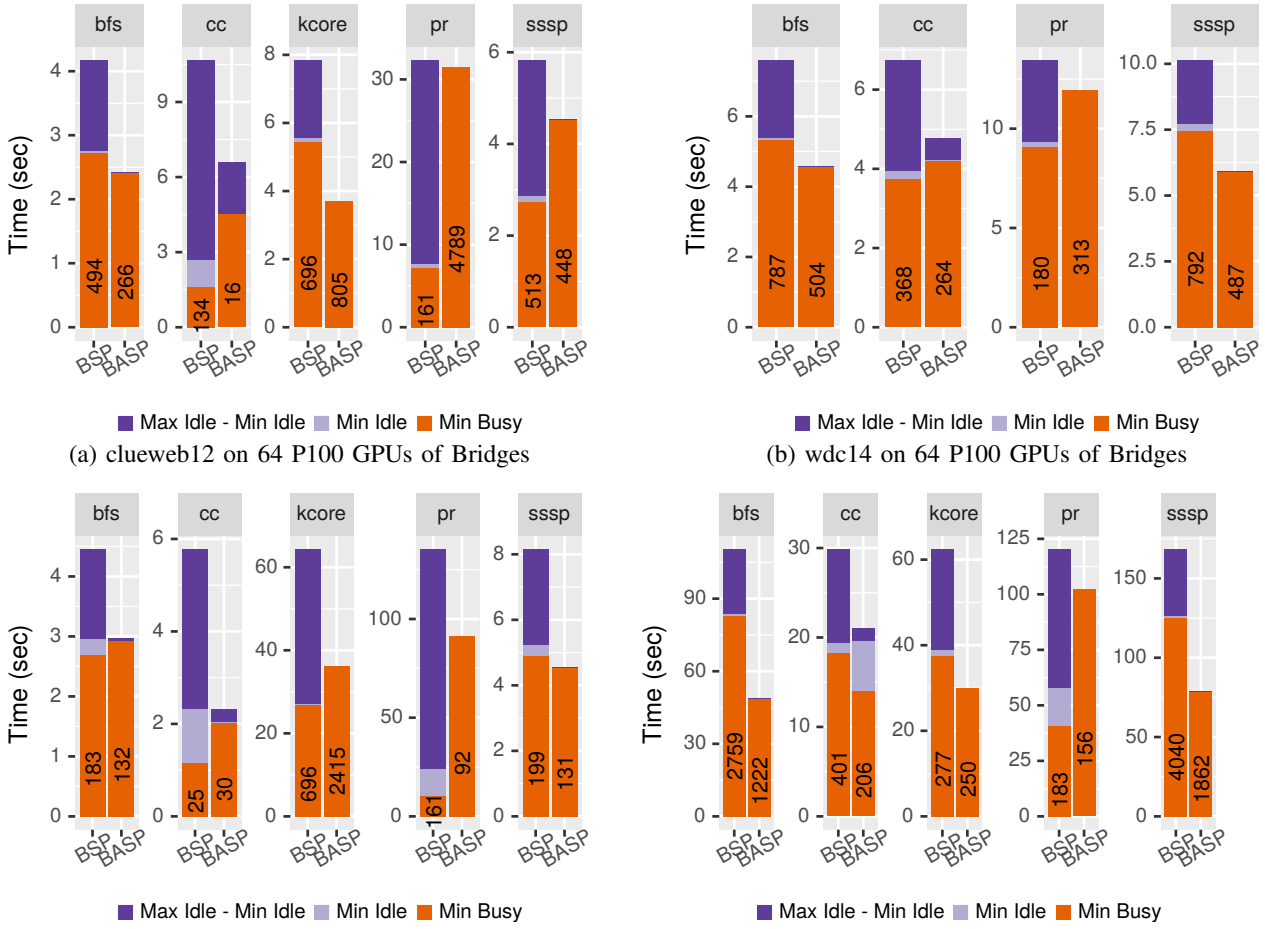
Fig. 10: Load imbalance in Gluon-Sync (presented as relative standard deviation in computation times among devices).

plot[5] for all the number of devices (CPUs or GPUs) for each benchmark and input graph on Bridges and Stampede, respectively. Each point in a box-plot is a value for a distinct configuration of the number of devices (CPUs or GPUs) for that benchmark and input graph. The load imbalance and the number of rounds can be used to tell whether Gluon-Sync can benefit from switching to BASP-style execution. As cc on gsh15 is well balanced and executes very few rounds, it does not benefit much from BASP-style execution. In contrast, benchmarks using clueweb12 are more imbalanced and benefit significantly from BASP-style execution, even if it executes very few rounds like in cc. For high-diameter graphs, load balance is difficult to achieve in efficient *data-driven* graph applications [11] because different subsets of nodes may be updated in different rounds. We show that Gluon-Async circumvents this by using BASP-style execution.

Idle time: We define busy time of a host as the time spent in computation, serialization (for packing messages to be sent), deserialization (for unpacking and applying received messages), and communication between host and device. The rest of the total time is the idle time; in BASP, idle time includes the time to detect termination. Different hosts can have different busy and idle times (stragglers have smaller idle times), so we consider the minimum and maximum across hosts. Figure 11 show the breakdown of execution time into minimum busy time, minimum idle time, and the difference between maximum and minimum idle time. As expected, BSP has high maximum idle time due to load imbalance and BASP reduces idle time, which is one of the main advantages of having bulk-asynchronous execution. However, this reduction in idle time could lead to a corresponding increase in busy time because the host could be doing redundant or useless *work* by operating on stale values instead of being idle. This depends on the input graph and the benchmark. In some cases like pr, the busy time increases even though the idle time is reduced. In most other cases, the busy time does not increase by much. Nevertheless, it is clear that the difference between BASP and BSP is in the idle time, and the total execution time will be reduced only if the idle time is reduced without an excessive increase in busy time.

Rounds executed: All hosts execute the same number of rounds in BSP (Table IV), whereas different hosts may execute different numbers of local rounds in BASP. The minimum

[5]The box for an input graph and benchmark represents the range of 50% of these values for that input graph and benchmark; the line dividing the box is the median of those values and the circles are outliers.



(a) clueweb12 on 64 P100 GPUs of Bridges

(b) wdc14 on 64 P100 GPUs of Bridges

(c) clueweb12 on 128 hosts of Stampede

(d) wdc12 on 128 hosts of Stampede

Fig. 11: Breakdown of execution time (sec); the minimum number of rounds executed among hosts is shown on each bar.

rounds executed in BSP and BASP are shown on each bar in Figure 11. We use the minimum local rounds among hosts to estimate the critical path in the execution. We count the number of edges processed (locally) on each host and use the maximum among hosts to estimate the work done in the execution. Figure 12 presents the correlation between the speedup in execution time, the increase or growth in the work done (maximum local work items or edges processed), and the reduction in the critical path (minimum local rounds). Each point is a value for a distinct configuration of benchmark, input, and number of devices (CPU or GPU); we have omitted outliers. Red (closer) points have lower growth in the work done and higher points (taller lines) have more reduction in the critical path. If BASP reduces both the work done (growth < 1) and the critical path (reduction > 1), then it would obviously be faster. As shown in the figure, BASP is faster than BSP (speedup > 1) when work done is reduced. More importantly, BASP does more work than BSP in many cases, but it is faster due to a reduction in the critical path. When BASP is slower than BSP (speedup < 1), it is due to a high growth in work done without sufficient reduction in critical path. Although the minimum number of local rounds in BASP may be smaller than that of BSP, the maximum number of local rounds in BASP may be higher because the faster hosts need not wait and may execute more local rounds. Instead of waiting after every round in BSP, faster hosts in BASP may execute more rounds. Consequently, faster hosts could make more progress and send updated values to the stragglers or slower hosts. The straggler hosts receive these updated values before they move to the next round, saving them from doing redundant work using stale values. Thus, straggler hosts doing fewer local rounds leads to faster convergence in BASP.

E. Summary and Discussion

Table V compares the performance of Gluon-Sync and Gluon-Async using the best-performing number of CPUs and GPUs. Both Gluon-Sync and Gluon-Async mostly scale well, so their best performance is usually on the maximum number of CPUs or GPUs we evaluated. For low-diameter graphs, Gluon-Async and Gluon-Sync are comparable. For high-diameter graphs, Gluon-Async is on average $\sim 1.5\times$ faster than Gluon-Sync. The speedup varies depending on the

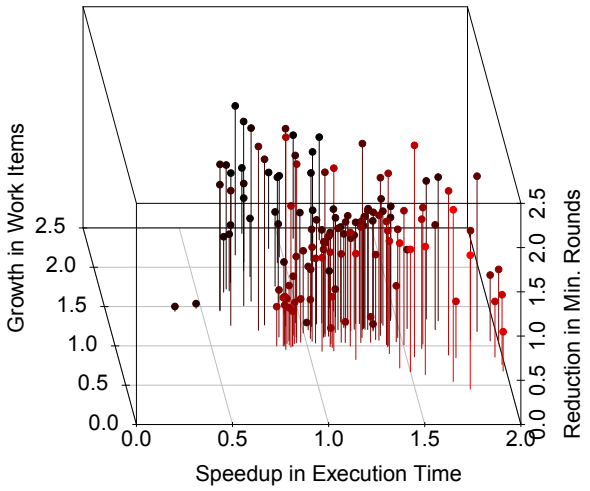


Fig. 12: BASP over BSP: correlation between speedup, growth in maximum # local work items, and reduction in minimum # local rounds for all benchmarks, inputs, and devices (CPUs or GPUs). Red color indicates lower growth in work items.

benchmark, the input, and the scale (number of devices). The speedup is typically best for high-diameter graphs at scale. This is similar to what has been observed for asynchronous execution on CPUs [23] or GPUs [40]. Thus, Gluon-Async helps scaling out large real-world graph datasets.

V. Related Work

Asynchronous Distributed Graph Analytics Systems. The popularity of the bulk-synchronous parallel (BSP) model [10] of computation has led to work that improves its performance by improving the underlying asynchrony and reducing the wait time. GraphLab [12] and PowerSwitch [14] systems use their gather-apply-scatter model along with distributed locking for non-blocking, asynchronous execution. None of the other systems, including Gluon-Async, use locks. Systems like Aspire [13], GRACE [15], Giraph++ [16], and ASYMP [41], which are based on asynchronous parallel (AP) model, avoid delaying the processing of the already arrived messages. GiraphUC [42] proposes the barrierless asynchronous parallel (BAP) model that uses local barriers to reduce the message "staleness" and overheads due to global synchronization. While GiraphUC is lock-free and asynchronous, it blocks during synchronization until it receives the first message (from any host). Most recently, Fan et al. [17] show that the Adaptive Asynchronous Parallel (AAP) model used in their GRAPE+ system can be used to dynamically adjust the relative progress of different worker threads and reduce the stragglers and stale computations. Similarly, Groute [40] proposes an asynchronous system, but it is limited to a single node system.

Most of these existing systems either perform fine-grained synchronization or do not support general partitioning policies.

TABLE V: Fastest execution time (sec) of Gluon-Sync and Gluon-Async using the best-performing number of hosts (# of hosts in parenthesis; "-" indicates out of memory; best among Gluon-Sync and Gluon-Async in bold and highlighted).

Bench-mark	Input	CPUs (Stampede)		GPUs (Bridges)	
		Gluon-Sync	Gluon-Async	Gluon-Sync	Gluon-Async
bfs	gsh15	1.3 (128)	**0.8 (128)**	0.9 (64)	**0.7 (64)**
	clueweb12	4.5 (128)	**3.0 (128)**	4.2 (64)	**2.4 (64)**
	uk14	13.0 (128)	**8.8 (128)**	8.8 (64)	**7.4 (64)**
	wdc14	9.3 (128)	**6.5 (128)**	7.6 (64)	**4.6 (64)**
	wdc12	110.3 (128)	**48.9 (128)**	-	-
cc	gsh15	**1.0 (128)**	**1.0 (128)**	1.2 (64)	**1.1 (64)**
	clueweb12	5.8 (128)	**2.3 (128)**	10.7 (64)	**6.6 (64)**
	uk14	2.2 (64)	**1.3 (128)**	10.4 (64)	**6.1 (64)**
	wdc14	7.3 (128)	**5.2 (128)**	6.7 (64)	**4.8 (64)**
	wdc12	29.8 (128)	**21.0 (128)**	-	-
kcore	gsh15	9.8 (128)	**6.9 (128)**	3.0 (64)	**2.3 (64)**
	clueweb12	64.3 (128)	**36.2 (128)**	7.8 (64)	**3.7 (64)**
	uk14	11.8 (128)	**6.4 (128)**	2.2 (64)	**1.7 (64)**
	wdc14	18.4 (128)	**9.4 (128)**	-	-
	wdc12	62.4 (128)	**29.9 (128)**	-	-
pr	gsh15	47.3 (128)	**46.2 (64)**	14.0 (64)	**10.5 (64)**
	clueweb12	130.5 (16)	**91.6 (128)**	32.3 (64)	**24.9 (32)**
	uk14	11.7 (128)	**10.1 (128)**	6.3 (64)	**5.6 (64)**
	wdc14	**24.7 (128)**	25.3 (128)	13.4 (64)	**11.9 (64)**
	wdc12	120.2 (128)	**102.2 (128)**	-	-
sssp	gsh15	2.9 (128)	**1.9 (128)**	2.8 (64)	**2.1 (64)**
	clueweb12	8.1 (128)	**4.6 (128)**	5.1 (32)	**4.0 (32)**
	uk14	16.3 (128)	**13.0 (128)**	12.4 (64)	**9.0 (64)**
	wdc14	10.9 (128)	**7.7 (128)**	10.1 (64)	**5.9 (64)**
	wdc12	168.3 (128)	**78.9 (128)**	-	-

None of them can be extended for vertex-cuts without significantly increasing the communication cost; i.e., some of the communication optimizations [5] would need to be dropped for such an extension (to elaborate, GRAPE+ is the only one that can support vertex-cuts without using distributed locks, but they send an updated value from a proxy directly to all the other proxies instead of reducing updated values to a master proxy and broadcasting the result to mirror proxies, resulting in more communication volume and messages). Consequently, prior asynchronous systems do not perform as well as the state-of-the-art BSP-style distributed systems [4], [5]. Moreover, none of the prior asynchronous systems can be extended trivially to support execution on multi-host multi-GPUs.

In contrast, we propose a Bulk-Asynchronous Parallel (BASP) model for both distributed CPUs and GPUs in which the threads potentially never wait and instead continue to do local work if available without explicitly waiting for the communication from other hosts. Our redesign of reduce and broadcast communication phases enables removing synchronization while exploiting bulk-communication.

Bulk-Synchronous Distributed Graph Analytics Systems. There have been many works that support graph analytics on distributed CPUs [2], [3], [4], [5], [43] or GPUs [5], [6] in the Bulk-Synchronous Parallel (BSP) model. Our proposed approach targets wait-time reduction in graph applications by exploiting the underlying asynchrony in codes written in BSP models, and it targets distributed CPU and GPU systems.

VI. Conclusion

This paper presented a novel programming model called BASP that takes bulk-communication from BSP models and continuous compute from asynchronous models to improve overall runtime of programs. We showed that it is easy to adapt BSP programs for BASP execution by modifying programs in D-Galois and D-IrGL, the state-of-the-art distributed graph analytics systems for CPUs and GPUs, respectively. Both these systems use the Gluon substrate for communication. We modified Gluon to support BASP and build the first asynchronous distributed and heterogeneous graph analytical system, Gluon-Async (source code is publicly available [27]). Gluon-Async retains the benefits of Gluon, so it can handle arbitrary partitioning policies and can be used to extend existing CPU or GPU graph analytical systems for distributed and heterogeneous execution. Our evaluation shows that on real-world large-diameter graphs at scale, BASP programs are on average $\sim 1.5\times$ faster than D-Galois and D-IrGL, respectively. Gluon-Async also scales better than them.

Acknowledgments

This research was supported by the NSF grants 1406355, 1618425, 1705092, 1725322 and by the 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] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel Graph Analytics," *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2901919

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings ACM SIGMOD Intl Conf. on Management of Data*, ser. SIGMOD '10, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387883

[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A Computation-centric Distributed Graph Processing System," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 301–316. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026901

[5] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '18. New York, NY, USA: ACM, 2018, pp. 752–768. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192404

[6] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *Proc. VLDB Endow.*, vol. 11, no. 3, pp. 297–310, Nov. 2017. [Online]. Available: https://doi.org/10.14778/3157794.3157799

[7] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[8] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 646–655.

[9] E. G. Boman, K. D. Devine, and S. Rajamanickam, "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)*, Nov 2013, pp. 1–12.

[10] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The TAO of parallelism in algorithms," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. PLDI '11, 2011, pp. 12–25. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993501

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proceedings VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354

[13] K. Vora, S. C. Koduru, and R. Gupta, "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*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 861–878. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660227

[14] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 194–204. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688508

[15] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, vol. 13, 2013, pp. 3–6.

[16] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "Think Like a Vertex" to "Think Like a Graph"," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, Nov. 2013. [Online]. Available: http://dx.doi.org/10.14778/2732232.2732238

[17] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu, "Adaptive Asynchronous Parallelization of Graph Algorithms," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 1141–1156. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196918

[18] T. A. Kanewala, M. Zalewski, and A. Lumsdaine, "Families of Graph Algorithms: SSSP Case Study," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 428–441.

[19] J. S. Firoz, M. Zalewski, A. Lumsdaine, and M. Barnas, "Runtime Scheduling Policies for Distributed Graph Algorithms," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 640–649.

[20] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, "CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics," in *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2019, 2019.

[21] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms," ser. PVLDB, vol. 12, no. 4, 2018.

[22] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 249–264.

[23] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 456–471. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522739

[24] L. Hoang, M. Pontecorvi, R. Dathathri, G. Gill, B. You, K. Pingali, and V. Ramachandran, "A round-efficient distributed betweenness centrality algorithm," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 272–286. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295729

[25] A. Kshemkalyani and M. Singhal, *Distributed computing: Principles, algorithms, and systems*. Cambridge University Press, 2008.

[26] H.-V. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir, "A Lightweight Communication Runtime for Distributed Graph Analytics," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[27] "Galois system," 2019. [Online]. Available: http://iss.oden.utexas.edu/?p=projects/galois

[28] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, "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*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 30:1–30:8. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792775

[29] "Pittsburgh Supercomputing Center," 2019. [Online]. Available: https://www.psc.edu/

[30] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science and Engineering*, vol. 16, no. 5, pp. 62–74, Sept-Oct 2014.

[31] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "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*, ser. PEARC17. New York, NY, USA: ACM, 2017, pp. 15:1–15:8. [Online]. Available: http://doi.acm.org/10.1145/3093338.3093385

[32] P. Boldi and S. Vigna, "The WebGraph framework i: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[33] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 587–596. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963488

[34] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[35] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, pp. 442–446. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[36] T. L. Project, "The ClueWeb12 Dataset," 2013. [Online]. Available: http://lemurproject.org/clueweb12/

[37] P. Boldi, A. Marino, M. Santini, and S. Vigna, "Bubing: Massive crawling for the masses," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014, pp. 227–228. [Online]. Available: http://doi.acm.org/10.1145/2567948.2577304

[38] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Web data commons - hyperlink graphs," 2012. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[39] S. Pai and K. Pingali, "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*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984015

[40] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "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*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 235–248. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018756

[41] E. Fleury, S. Lattanzi, V. S. Mirrokni, and B. Perozzi, "ASYMP: fault-tolerant mining of massive graphs," *CoRR*, vol. abs/1712.09731, 2017. [Online]. Available: http://arxiv.org/abs/1712.09731

[42] M. Han and K. Daudjee, "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 950–961, May 2015. [Online]. Available: https://doi.org/10.14778/2777598.2777604

[43] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741970