SHMEMGraph: Efficient and Balanced Graph Processing Using One-sided Communication

Huansong Fu* Manjunath Gorentla Venkata† Shaeke Salman* Neena Imam† Weikuan Yu*
Florida State University* Oak Ridge National Laboratory†
{fu,salman,yuw}@cs.fsu.edu {manjugv,imamn}@ornl.gov

***Abstract*—State-of-the-art synchronous graph processing frameworks face both *inefficiency* and *imbalance* issues that cause their performance to be suboptimal. These issues include the inefficiency of communication and the imbalanced graph computation/communication costs in an iteration. We propose to replace their conventional two-sided communication model with the one-sided counterpart. Accordingly, we design *SHMEMGraph*, an efficient and balanced graph processing framework that is formulated across a global memory space and takes advantage of the flexibility and efficiency of one-sided communication for graph processing. Through an efficient one-sided communication channel, SHMEMGraph utilizes the high-performance operations with RDMA while minimizing the resource contention within a computer node. In addition, SHMEMGraph synthesizes a number of optimizations to address both *computation imbalance* and *communication imbalance*. By using a graph of 1 billion edges, our evaluation shows that compared to the state-of-the-art Gemini framework, SHMEMGraph achieves an average improvement of 35.5% in terms of job completion time for five representative graph algorithms.**

I. Introduction

Graph processing has emerged as a very attractive practice for data analytics. As the size of graph data keeps exploding, they become increasingly hard to be fit into a single machine [26], [8]. Therefore, distributed graph processing frameworks become the main focus for large-scale graph processing [21], [20], [10], [25], [24], [26], [7], [28]. In particular, Gemini [28] is a recent effort whose computation-centric design has improved previous works by at least an order of magnitude.

However, the existing distributed and synchronous graph processing frameworks still suffer from serious inefficiency and imbalance issues. First of all, despite a few early attempts, the state-of-the-art graph processing frameworks are still unable to efficiently utilize the high-performance interconnects and capabilities such as Remote Direct Memory Access (RDMA). Moreover, there exists an imbalance issue in synchronous graph processing that renders its performance to be suboptimal. This problem arises from two sources: *computation* and *communication*. For example, in an iteration of the Bulk-Synchronous Parallel (BSP) model, different pairs of nodes have uneven computation and communication needs and therefore have unmatched progresses. This creates idleness for certain nodes and delays the completion of the current iteration. The cause for this issue is convoluted and hard to be eradicated. However, our investigation indicates that, if we can relax the strict synchronization barrier imposed by the two-sided communication and leverage its one-sided counterpart, we can greatly reduce the node idleness and delays.

To this end, we propose SHMEMGraph, a more efficient and balanced graph processing framework that uses one-sided operations to exchange remotely-accessible graph data between different nodes in a shared-memory style [1]. Several benefits comes with such a design. Firstly, the memory-style addressing matches up the distributed in-memory graph processing better than the conventional view. Secondly, the one-sided operations that are used in the shared memory model can improve the communication efficiency of distributed graph processing, especially with RDMA. Thirdly, by leveraging the flexibility of one-sided operations, the aforementioned imbalance issues can be mitigated through more flexible and fine-grained load balancing, which is prohibitively difficult with the traditional two-sided communication.

In this paper, we will show SHMEMGraph's design over the state-of-the-art Gemini framework [28] as proof-of-concept. SHMEMGraph addresses the inefficiency and imbalance issues by incorporating several novel techniques. First of all, its basic communication channel uses only one single thread to conduct one-sided `Put` for transferring data, which is more efficient and scalable than the existing solutions. Secondly, we divide the long and imbalanced computation process into finer-grained pieces so that they can be better overlapped and cause less node idleness. Thirdly, we further exploit the flexibility of one-sided `Get` to implement a communication balancing mechanism to further reduce the delaying time.

In this paper, we first briefly cover the background in Section II. Then we present the motivation and design of our work in Section III and IV. After that, We show evaluation results in Section V and discuss related works in Section VI.

II. Background

A. Distributed Graph Processing

In this paper, we focus on vertex-centric graph processing, where a graph consists of many vertices connected by directed/undirected edges. Here, we briefly describe the basic *execution* and *programming* models of the existing distributed graph processing frameworks.

[1]The acronym SHMEM, in this context, represents SHared MEMory rather than Symmetric Hierarchical MEMory although we have also used OpenSHMEM [6] for SHMEMGraph.

Synchronous (or *bulk-synchronous*) and *asynchronous* are two major execution models for graph processing. They are distinguished by whether we put a synchronization barrier between each *iteration* of processing. Although the asynchronous model poses no global synchronization barrier, it often incurs costly lock contention and lacks message batching. As a result, its performance is generally less competitive and sometimes unable to ensure correct convergence for certain graph algorithms [12], [26], [28]. Most of the existing graph processing frameworks support either both models (e.g. [20], [10], [24]), or only the synchronous one (e.g. [21], [11], [28]). In general, each iteration in the synchronous model consists of three phases: *preparation*, *communication* and *processing*. Each phase is conducted in parallel on different nodes and can be pipelined to allow overlapping with other phases.

There are various programming models for writing graph algorithms, but they generally involve a *vertex program* which defines the way to prepare, communicate and process vertices/edges. Depending on the direction of initiating the communication along the edges, most of the programming models are either *push*-based (source vertices conduct computation and send updates to destination vertices) [21], [26], [18], *pull*-based (destination vertices fetch source data and update themselves) [20], [10], [11], or hybrid push/pull [25], [28], [3]. Although every synchronous graph processing framework may have varying amount of work for each preparation/communication/processing phase, it is universally important to balance each node's cost in the same phase in order to achieve good parallelism and less resource idling.

B. Gemini

Gemini [28] is a distributed and synchronous graph processing framework with a computation-centric design. Gemini uses a low-overhead edge-cut partitioning strategy to distribute graph data, which balances each node's workload by considering the number of vertices and the number of outgoing edges originated from those vertices. In addition, Gemini's hybrid model adaptively chooses either dense mode (similar to push) or sparse mode (similar to pull) in each iteration, based on the current graph density (i.e. number of *active* vertices). Gemini also provides NUMA awareness, making worker threads to primarily work on local NUMA nodes and secondarily on remote ones. On top of that, Gemini's inter-node communication is conducted using MPI send/receive operations in a round-robin fashion, where every node sends data to another node in a predetermined order. In each round (only conceptually, no synchronization barrier is enforced), each node sends data to a certain node and receives data from another. Gemini greatly mitigates the computation bottleneck and has good overlapping between computation and communication. As a result, it achieves orders of magnitude of performance improvement compared to its prior works.

III. Inefficiency and Imbalance of the Existing Framework

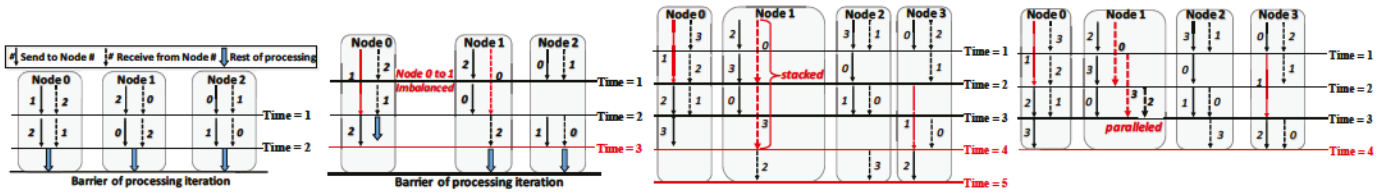
In this section, we will use the state-of-the-art Gemini framework to investigate the imbalance problem and its overall job performance impact. In our experiment, we run PageRank for 10 iterations on the twitter-2010 graph on 4 nodes, which is a typical workload used by many previous works to study graph processing problems [16], [23], [28]. Note that, the problems are not limited to a few applications in a specific framework, they are due to fundamental limitations of synchronous graph processing, partitioning strategies, etc. The detailed experimental setup can be found in Section V.

A. Communication Inefficiency

For many graph applications, the processing of graph data usually generates a large amount of data to be exchanged between different nodes. Although many graph processing frameworks propose various techniques to overlap the communication and computation, slow communication can often result in a large portion of time spent in communication and thus prolongs the job completion time. For example, when running small graphs (e.g. enwiki-2013, with around 100 million edges), Gemini spends a dominant portion of time on communication [28]. In that case, the efficiency of the communication will largely determine the job performance. For larger graphs, the communication costs can be better hidden behind computation, but the efficiency of communication still matters because the graph often becomes sparser after multiple iterations and the computation cannot hide the communication well anymore. Although advanced network capability such as Remote Direct Memory Access (RDMA) has shown to be able to significantly boost the communication efficiency for many distributed systems,unfortunately however, the state-of-the-art graph processing frameworks have not yet taken full advantage of it despite a few early attempts which adopt a less-efficient communication channel design (discussed in Section VI).

B. The Imbalance Problem

In the synchronous model of graph processing, every node synchronizes at the end of a processing iteration. Therefore, the slowest node to complete will determine the time duration for the current processing iteration. Ideally, in each iteration, the same parallel portion at each node costs about the same, as shown in Fig. 1(a). In that case, every node will synchronize roughly at the same time. There is very little idleness and the parallelism gets maximized. However, in real world, the progresses of certain nodes are often imbalanced. A particular node can be delayed, which further delays the completion of the current processing iteration and the overall job performance. Fig. 1(b) shows an example, where the data sending/receiving from node 0 to node 1 gets delayed. As shown in the figure, compared to the ideal case, the delay creates idleness on node 2 and prolongs the completion of the current processing iteration by the same amount. Such a *node-to-node imbalance* has largely been overlooked by prior arts whose main focus has been the overall imbalance in a whole



(a) The ideal case. (b) The imbalanced case. (c) The combined effect with two stacked delays. (d) The suboptimal solution to the combined effect.

Fig. 1: Illustration of the imbalance problem in graph processing.

processing iteration [10], [7]. Note that, For a node that causes imbalance, its faster round may not remedy its slower round since the delay also affects another node.

Even worse, however, there is a **combined effect of imbalance** where the slower rounds from different nodes are affecting the job performance altogether. As illustrated in Fig. 1(c), the data transferring from node 0 to node 1 and the one from node 3 to node 1 both get delayed. As a result, the two delays are stacked together at node 1, adding twice as much delays to the completion of the iteration. In Gemini, this combined effect is avoided by launching many receiving threads on every node. Each thread receives data from one of the other participating nodes, as shown in Fig. 1(d). However, this approach is suboptimal as it is both costly and incomplete: firstly, each node needs to launch the same number of threads as the total number of participating nodes, making the system harder to scale out; secondly, as shown in Fig. 1(d), one of the delays is still carried over to the final completion, leaving the performance still being affected.

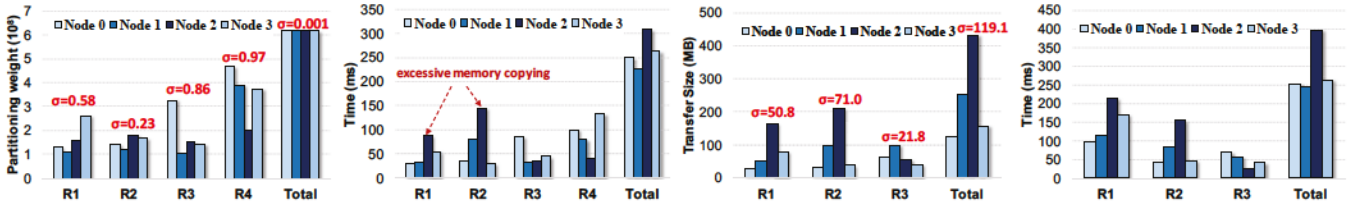
1) Causes of Imbalance: In general, the delay shown in Fig 1(b) can be originated from the preparation or communication phases as mentioned before. We refer to these two types of imbalance as *computation imbalance* and *communication imbalance*. The processing phase has much less impact in the case of Gemini due to much less computation work to do, though it may has a larger imbalance issue in another framework. Next, we study the reasons behind the two major types of imbalance, along with cost analysis.

Computation Imbalance: Although most graph partitioning strategies make their best efforts in evenly distributing graph data among the nodes, significant computation imbalance still exists. Many reasons are accountable for that. First and most importantly, a common partitioning strategy can only balance the total computation cost of each node in each *iteration*, but not the computation costs of them in each *round*. Take Gemini's partitioning strategy for example, it partitions the graphs based on each node's *partitioning weight* that is calculated from the number of a node's owned vertices and their outgoing edges. As shown in Fig. 2(a), where we plot the partitioning weights for different node-to-node computation work in each round in our motivating experiment, the total partitioning weights for different nodes in the experiment are well balanced. However, the partitioning weights in each round are imbalanced drastically (as seen in the variation σ). The reason is that, the partitioning strategy cannot coordinate the amount of connected edges for every pair of nodes, thus being unable to balance the partitioning weights in each round. The cost analysis of this issue in shown in Fig. 2(b). As we compare Fig. 2(a) with Fig. 2(b), large partitioning weights are directly associated with the long computation costs. There are some significant nonconformity though, which is due to significant larger number of outgoing edges that Node 2 owns for Node 0 and 3 (but it owns much less vertices so its partitioning weight is evened out). Therefore, it needs more memory accesses for writing the data to message buffer, which makes its computation cost higher than expected.

Communication Imbalance: Similar to the computation imbalance, the communication imbalance stems from the uneven node-to-node communication needs. The imbalanced numbers of edges that each node connects to another node will generate uneven amount of sending data among them in each round, which is hard to solve using the existing partitioning strategies. In addition, the outcome of the computation will have an effect on communication imbalance since each node can generate different amount of sending data, depending on the algorithm, graph topology etc. Fig. 2(c) shows the sizes of transferring data in each round and in total. We can see that the sizes vary significantly. Especially, the excessive number of outgoing edges on Node 2 has a huge impact on its communication need. Moreover, even the total communication costs of each node are not well balanced due to the unpredictability of the computation outcome.

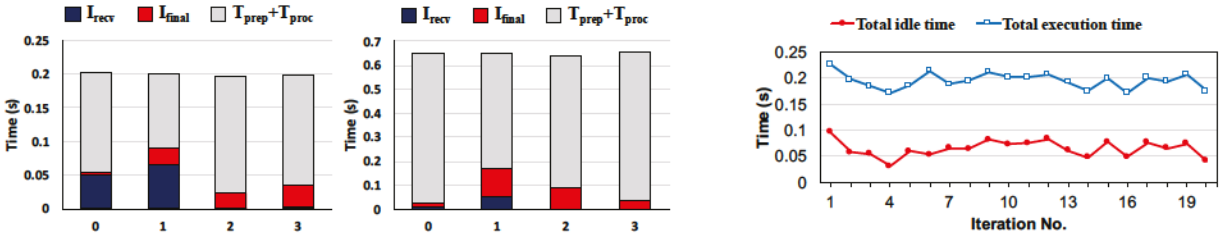
There are also other reasons for computation and communication imbalance, which contribute to the variations in Fig. 2(b) and Fig. 2(d) as well. For example, some input graph data do not have *continuous* numbers for vertices, i.e. a large portion of the vertices range do not appear in the edges. Simple edge-cut partitioning [28] may lead to a very biased distribution of vertices/edges for these graphs because it partitions vertices based on the total range of vertex numbers. Other common factors include local resource contention, network congestion, node heterogeneity etc.

2) Implications to Overall Job Performance: Till now, we have only shown the direct effect of the imbalance issue in each round, but not yet seen its implication to the overall job performance. It is hard to get things into perspective using the job completion time because there is hardly an ideal case without the imbalance issue that we can compare with. However, we find that since the imbalance issue causes delays by creating idleness for certain nodes, we can understand the overall cost and room for improvement by quantifying the *idle time*. To put it formally, without computation/communication



(a) Imbalanced partitioning weights in each round. (b) Imbalanced computation costs in each round. (c) Imbalanced transferring sizes in each round and in total. (d) Imbalanced communication costs in each round and in total.

Fig. 2: Cause and effect of computation and communication imbalance.



(a) Idle time on an in-house cluster. (b) Idle time on Titan. (c) Variation of idle time on Node 1.

Fig. 3: Overall performance impact of the imbalance problem.

overlapping, the time for each node to complete a processing iteration equals to $T_{prep} + T_{comm} + T_{proc} + I_{final}$, where T_{prep}, T_{comm} and T_{proc} are the times for preparation, communication and processing phases, respectively, and I_{final} is the idle time in waiting for other nodes at the end of iteration. Additionally, when communication is overlapped with computation, the completion time becomes $T_{prep} + I_{recv} + T_{proc} + I_{final}$, where T_{comm} is replaced by I_{recv}, which is the idle time in waiting for receiving data. Without further condensing or overlapping the T_{prep} and T_{proc} costs (also extremely difficult), a realistic goal for optimizations should aim for reducing each node's I_{recv} and I_{final} and also balancing T_{prep} and T_{proc} between different nodes as good as possible.

The first two figures in Fig. 3 have shown the two types of idle time of each node for our motivating experiment on two systems, which have very different computation and communication costs for Gemini (detailed discussion can be found in Section V). From the figure, we can see that for both systems, the imbalance problem has caused large and skewed idle times, which have seriously affected the overall job performance. In addition, compared to in-house cluster, the idle time is less pronounced on Titan because the computation takes more time there so better computation/communication overlapping has been made possible.

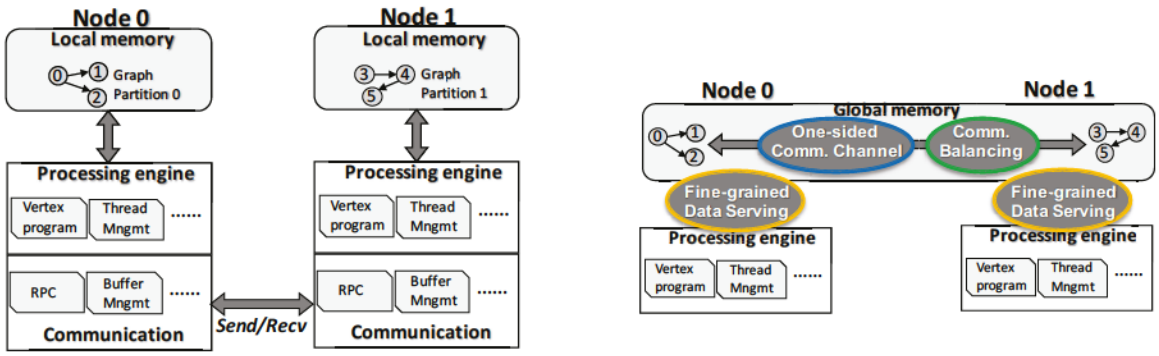
The third figure (Fig. 3(c)) shows the relation between total idle time of the most affected node (Node 1) and total execution time of each iteration. We run 20 iterations of PageRank to show it at a longer time-span. We can see that the two lines have similar trend, which shows the correlation of the idle time and iteration's execution time.The very few occasions where the two lines do not match (e.g. Iteration 6) is when the node itself has encountered additional certain delays so it is much less affected by other nodes in that iteration. Also, both lines are highly fluctuating due to random system/network overheads.Such performance variation has made designing countermeasures even more challenging because even if we can eliminate the major causes for imbalance, we still need to deal with the variations in order to provide more stable and predictable performance for graph processing applications.

IV. Design of SHMEMGraph

In this section, we present the design of SHMEMGraph, a graph processing framework that cleverly leverages the efficiency and flexibility of one-sided operations to mitigate the negative performance impact of aforementioned issues.

A. Overview

The overarching goal of SHMEMGraph is to support various distributed and synchronous graph processing frameworks and provide more balanced and efficient computation and communication for them. Fig. 4 shows the comparison between the architectures of traditional graph processing frameworks and SHMEMGraph. The traditional one (Fig. 4(a)) shows the most representative architecture of the existing distributed graph processing frameworks, where each node processes their graph partitions stored in local memory and uses certain types of RPC methods (even with RDMA) to transfer graph data or computation results to other nodes. In contrast, SHMEMGraph (Fig. 4(b)) has three new/modified components for achieving its design goals. Firstly, the basic one-sided communication channel in SHMEMGraph conducts one-sided operations on the global memory space that stores graph data, computation output etc. The basic channel uses a single thread to remotely write data to other nodes using one-sided `Put`. This facilitates the efficient utilization of RDMAand also resolves the combined effect of the imbalance issue at a much lower cost. Secondly, SHMEMGraph breaks some overlong phases of preparation, communication and processing all into finer-grained pieces. This *fine-grained data serving* mechanism helps SHMEMGraph reduce the delay resulting from computation imbalance. Thirdly, SHMEMGraph further leverages the one-sided `Get` to implement a *communication*



(a) Traditional. (b) SHMEMGraph.

Fig. 4: Architectures of traditional graph processing frameworks and SHMEMGraph.

balancing mechanism, which counters the performance impact of communication imbalance and saves unnecessary idle time in waiting for the receiving data.

Note that, such a design is effective for any graph programming model (i.e. push, pull or hybrid) as they all have similar interdependent preparation, communication and processing phases. The actual amount of work in each phase may differ, which will determine how much a certain framework will benefit from our techniques. For example, for some frameworks that have very few or no processing work after communication [21], [15], the fine-grained data serving will be unable to provide much improvement, whereas the communication balancing will be more effective due to larger portion of communication work in an iteration. In addition, the basic communication channel of SHMEMGraph is especially helpful for systems that have the combined effect of imbalance (e.g. Gemini), but can still provide improvement for any other graph processing frameworks that need to send batch messages between all participating nodes (true for most cases).

B. One-sided Communication Channel

The basic design of SHMEMGraph's one-sided communication channel is shown in Fig. 5. For each node, only one communication thread is launched, which uses `Put` to directly write data to the exposed *receiving buffer* on other nodes. The exposed memory also holds the *sending buffers* and *length arrays*, which will be discussed later. Same as Gemini, the exposed buffer is associated with certain NUMA node so that both local reads and writes to those buffers are local to their corresponding NUMA nodes. We favor `Put` over `Get` here because `Put` involves lighter synchronization overheads: it has only one additional call for the buffer locking apart from `Put`, whereas for `Get`, there are more calls for purposes such as repetitive availability inquiry, buffer locking/unlocking, receiving notification besides the actual `Get`.

After the preparation work is done and the sending buffer is filled, the sender thread conducts an atomic Compare-And-Swap (CAS) operation to the corresponding length array on the receiver node. This single CAS serves three purposes: notifying about data arrival, informing about data size and locking the receiving buffer. If there is no conflict going on, the CAS writes the data length to corresponding location in the length array. The conflict is when another CAS has modified the same location before. The receiver checks the length array to get informed about both the arrival and length of the receiving data. The receiving data has a completion flag at the end of the valid data portion to confirm the completion of transferring. Note that, although each receiving buffer is uniquely associated with a certain sender, avoiding race condition using locks is still necessary due to the communication balancing, which will be discussed later.

To maximize computation/communication overlapping, SHMEMGraph also adopts a round-robin inter-node communication procedure similar to the original Gemini. However, SHMEMGraph conducts the processing phase out-of-order in order to avoid the combined effect of imbalance. The receiver node does not block itself waiting for data from an anticipated node. Instead, when the receiver does not get data from the buffer for the anticipated node, it continues to check the subsequent buffers. Upon data being received, the receiver node continues to do processing and will skip the corresponding receiving buffer the next time. If the data from the most anticipated node is processed, it will mark the next unprocessed node as the new anticipated node.

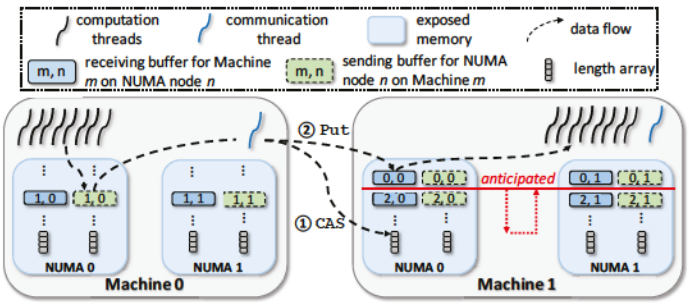


Fig. 5: SHMEMGraph's basic one-sided communication channel using a single thread.

C. Fine-grained Data Serving

On top of the basic one-sided communication channel, SHMEMGraph further tackles the computation imbalance by dividing the imbalanced and long data serving procedure of preparation, communication and processing phases all into finer-grained pieces. As shown in Fig 6, in this particular round, the sending data from Node 0 to Node 3 is divided into

fine-grained chunks. Upon completion of preparation phase, each individual chunk will be (1) written to the sending buffer, (2) sent to the remote receiving buffer, and (3) processed by the remote node upon being received.

In principle, the data serving procedure between a certain pair of node is chosen to be fine-grained if there is significant computation imbalance in current round. In the case of Gemini, the condition is when (1) the partitioning weight of the corresponding send/receive pair significantly exceeds the partitioning weight of other pairs on the same node, **or** (2) the number of outgoing edges associated with the corresponding sending data significantly exceeds the average number in the current iteration. More specifically, the sending data from node n_1 to n_2 is to be fine-grained if:

$$\frac{W^{(n_1,n_2)}}{W^{(n_1,N)}_{avg}} \times \frac{E^{(n_1,n_2)}}{E_{avg}} > \delta \tag{1}$$

where $W^{(n_1,n_2)}$ represents the partitioning weight of the send/receive pair from node n_1 to n_2, $W^{(n_1,N)}_{avg}$ represents the average partitioning weight of all the pairs from Node n_1, defined as $W^{(n_1,N)}_{avg} = \sum_{i=0}^{i<N, i\neq n_1} W^{(n_1,i)}/(N-1)$. In addition, $E^{(n_1,n_2)}$ is the number of outgoing edges from n_1 to n_2 and E_{avg} is the average number of outgoing edges for any pairs of nodes, calculated as $E_{avg} = \sum_{i=0}^{i<N} E^{(i,N)}/N$. Then, δ is the threshold to determine how much we should tolerate for minor imbalance, which is set to 1.5 empirically. Both factors in the equation contribute equally to the decision as a lower computation cost may be negated by a higher one.

In addition, because we cannot calculate the weights and edges by creating another computation phase which will add significant overheads, we collect the values along with preparation phase and use them as the predicting values for the next iteration (E_{avg} is made available to all nodes at the end of current iteration). For the first iteration, we use the values collected in the initialization phase. We do not simply apply fine-grained mode for all cases because only the imbalanced computation will cause significant delays. Also, we find that aggressively dividing the data serving procedure can cause adverse performance behavior due to increased operation overheads and resource contention.

Moreover, the length array is expanded to a two-dimensional array when the fine-grained data serving is enabled. Each original array entry is further extended to an array of lengths. The size for the sub-array is determined by the total length of each buffer and the length of each chunk, calculated as S_{buffer}/S_{chunk}. The chunk size S_{chunk} is an important tuning parameter in SHMEMGraph and is set empirically to 4 MB. After writing a chunk, the sender will conduct CAS on the corresponding position in the length array on the receiver side. The sender without fine-grained data serving will write the total length to the head of the array. The receiver looks up the length array and process the received chunks afterwards. In order to reduce frequent switch between different buffers and the lookup cost, the receiver will wait for a short while and then switch to the next buffer if no data arrives. Each node maintains a number of indicators to keep track of the processed chunks. If the receiving buffer is not fine-grained, the indicator will be at either the head or the tail position of the array, indicating empty or full, respectively.

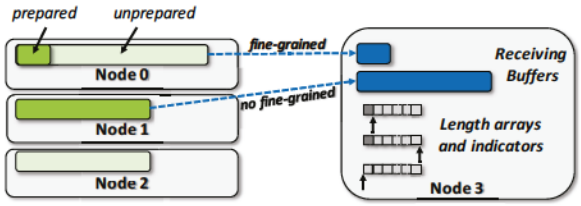


Fig. 6: Illustration of fine-grained data serving, where sending data from Node 0 to Node 3 is chosen to be divided.

D. Communication Balancing

The communication balancing mechanism allows the nodes that are faster in finishing their own sending data to speed up their data receiving progress. Based on the preparation progress on the sender node, two balancing approaches are used: fetching the sending buffer or fetching the unprocessed vertices. As shown in Fig 7, SHMEMGraph uses the one-sided `Get` for fetching data from other nodes. After a communication thread has finished sending all its data, it starts to remotely read data from another node that has not sent any data to this one. If it is currently receiving data from another node, i.e., the corresponding length array for that node has been locked, it will skip that node. Therefore, before fetching, the communication thread will also conduct a CAS to try to lock the corresponding length array entry. This prevents conflict in writing to the same buffer and avoids unnecessarily transmitting repetitive data. Similar to the basic communication channel, the receiver node will start fetching from its most anticipated sender node that has not yet sent data.

When balancing, a node first checks if the sending buffer on the target node has available data. It does that by checking the corresponding length array on the target node. Then the node fetches the corresponding data if it is available. If not, the node then checks if the active vertices on the target node can be fetched. The active vertices are prepared and stored in the global memory space by the target node at the beginning of each iteration. To avoid expensive communication overheads and shipped computation cost that may negate the benefit of communication balancing, we set a threshold s_T which is the upperbound for the number of active vertices that deserves to be fetched. In the case for Gemini, we set s_T to be $\frac{V_i}{10}$, where V_i is the number of total vertices that Node i owns. This value is chosen based on the general performance difference of sparse and dense modes in Gemini. Here, SHMEMGraph effectively exploits Gemini's adaptive mode switching within a single iteration: pull active vertices even in the push model.

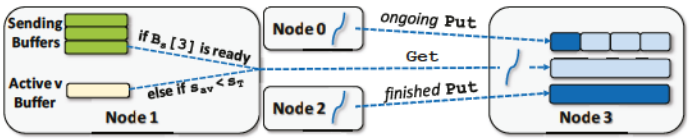


Fig. 7: Illustration of communication balancing.

V. Evaluation

We have fully implemented SHMEMGraph on top of Gemini using both C and C++. Since thread-safety is implementation-dependent and sometimes not provided by the one-sided communication libraries, we have also implemented a communication *delegator* [9] that runs alongside with the graph processing engine and uses shared memory to pass data between itself and the graph processing engine. We have managed to ship most of the techniques of the original SHMEMGraph to the delegator.

We have extensively evaluated the performance of SHMEMGraph against Gemini. Note that, we have confirmed on our testbed that both Gemini and SHMEMGraph can achieve more than an order of magnitude better performance over other distributed graph processing frameworks such as PowerGraph [10] and PowerLyra [7]. Such results are expected to be similar to the results reported in the Gemini's paper [28]. Due to space limitation, in this section we will only focus on the comparison between SHMEMGraph and Gemini to see how much SHMEMGraph can further improve the state-of-the-art Gemini framework. Next, we will summarize and analyze our experimental results in detail.

TABLE I: Graph data used in the experiments.

Name	Type	No. of vertices	No. of edges
enwiki-2013	continuous	4,206,289	101,311,614
twitter-2010	continuous	41,652,230	1,468,365,182
uk-2007-05	continuous	105,896,555	3,738,733,648
com-friendster	non-continuous	65,608,366	1,806,067,135

A. Experimental Setup

Unless otherwise specified, all experiments are conducted on an in-house cluster of 21 server nodes called *Innovation*. Each machine in Innovation cluster is equipped with 10 dual-socket Intel Xeon(R) cores and 64 GB memory. Hyperthreading is enabled to improve the graph computation performance. All nodes are connected through an FDR InfiniBand interconnect with the ConnectX-3 NIC. In addition, we have also evaluated SHMEMGraph on the *Titan* supercomputer at Oak Ridge National Laboratory [2]. Titan is a hybrid-architecture Cray XK7 system, which consists of 18,688 nodes and each node is equipped with a 16-core AMD Opteron CPU and 32GB of DDR3 memory.

We use Open MPI [1] v3.0.0 for both thread-safe (MPI) and non-thread-safe (OpenSHMEM) one-sided operations, whereas the recent OpenSHMEM specification 1.4 has also added thread safety feature. Note that, although Cray MPI is considered preferred on Titan, we only managed to get SHMEMGraph but not Gemini to work with it. Therefore, for Titan we use Open MPI to compare Gemini and SHMEMGraph but will show SHMEMGraph's performance using both Open MPI and Cray MPI (v7.6.3).

We use four representative graph workloads for our experiments, including three graphs with continuous vertices from WebGraph+LLP [5], [4] and a non-continuous graph from SNAP [17] (vertices range is 2x larger than actual number of vertices). The number of vertices and edges are shown in Table I. We evaluate five representative graph algorithms provided by the original Gemini, including PageRank (PR), Connected Components (CC), Single Source Shortest Path (SSSP), Betweenness Centrality (BC) and Breadth-First Search (BFS). For each result reported, we test the experiment at least 5 times and get the average. For results that we do not specify which setup is used, they are the results of running 10 iterations of PageRank on twitter-2010 on 4 nodes.

B. Overall Performance Improvement

First of all, we compare the overall job performance of Gemini and SHMEMGraph for each individual graph algorithm. As shown in Table II, SHMEMGraph is able to outperform Gemini for all test cases, including different graph algorithms, graph data type and testbed system. There are two general trends in these results. Firstly, for smaller graphs, SHMEMGraph has a larger improvement for algorithms that have a better portion of their processing iterations running with high graph density. Take PageRank for example, which does not change density throughout completion, it has the most significant improvement for enwiki-2013. This is because small graphs with low density have very little communication and computation works that also permit little room for improvement. However, for larger graphs, algorithms with more iterations running with low density see more improvement from SHMEMGraph. This is because at higher density, it takes much longer for computation than communication, resulting in less benefit from improved communication. The major performance improvement becomes the mitigation of the imbalance.

Moreover, in general, SHMEMGraph has a larger improvement on Innovation cluster than on the Titan supercomputer. This is because computation on Titan takes much longer due to less computation threads being used (16 compared to 40) and the lack of NUMA locality (Titan needs different preparation for thread-to-core binding). As a result, the communication for graphs of high density is completely hidden behind the long computation. Therefore, even communication balancing will not have much effect and SHMEMGraph can only benefit from the fine-grained data serving.

In addition, on Titan we have also evaluated SHMEMGraph using Cray MPI. As shown in Table III, where we run the five algorithms on the enwiki-2013 graph on 4 nodes, SHMEMGraph performs slightly better with Cray MPI on Titan. We have found that the main contributing factor here is the faster MPI RMA operations of Cray MPI over Open MPI on Titan. This is also reflected by the fact that algorithms with more communication work have larger improvement (e.g. 13.6% for PR but 6.7% for SSSP).

Finally, we can see that SHMEMGraph can also mitigate the imbalance caused by suboptimal partitioning of the non-continuous graph. However, those results are still not as good as twitter-2010 which has similar size. There is extremely long computation on certain node where a lot of vertices are converged. In that case, even our balancing techniques could not help either. This indicates that there is still need for a better partitioning strategy to deal with this type of graphs.

TABLE II: Comparison of job completion time (second) on 4 nodes.

Graph data: enwiki-2013											
System: Innovation						**System: Titan**					
	PR	CC	SSSP	BC	BFS		PR	CC	SSSP	BC	BFS
Gemini	0.43	0.29	0.44	2.25	1.19	Gemini	0.76	0.69	4.94	1.73	0.89
SHMEMGraph	0.24	0.21	0.34	1.91	1.05	SHMEMGraph	0.59	0.60	4.33	1.51	0.83
Improvement	44.2%	26.3%	23.1%	15.1%	11.8%	*Improvement*	22.4%	13.0%	12.3%	12.7%	6.7%
Graph data: twitter-2010											
System: Innovation						**System: Titan**					
	PR	CC	SSSP	BC	BFS		PR	CC	SSSP	BC	BFS
Gemini	2.45	1.32	2.79	1.64	0.74	Gemini	8.23	8.61	15.73	3.33	1.21
SHMEMGraph	1.9	0.86	1.73	1.03	0.48	SHMEMGraph	7.16	7.44	12.76	2.87	0.92
Improvement	22.4%	34.8%	38.0%	37.2%	35.1%	*Improvement*	13.0%	13.6%	18.9%	13.8%	24.0%
Graph data: uk-2007-05											
System: Innovation						**System: Titan**					
	PR	CC	SSSP	BC	BFS		PR	CC	SSSP	BC	BFS
Gemini	1.12	1.82	4.42	5.62	2.58	Gemini	4.44	5.82	28.97	9.31	3.65
SHMEMGraph	1.01	1.44	4.17	5.01	2.22	SHMEMGraph	4.07	5.09	27.14	8.63	3.23
Improvement	9.8%	23.4%	5.7%	10.9%	14.0%	*Improvement*	8.3%	12.5%	6.3%	7.3%	11.5%
Graph data: com-friendster											
System: Innovation						**System: Titan**					
	PR	CC	SSSP	BC	BFS		PR	CC	SSSP	BC	BFS
Gemini	7.21	4.79	9.75	2.22	0.91	Gemini	29.85	19.05	82.17	6.13	2.41
SHMEMGraph	6.13	4.33	8.63	1.91	0.68	SHMEMGraph	24.01	16.58	76.33	5.29	2.07
Improvement	15.0%	9.6%	11.5%	14.0%	25.3%	*Improvement*	19.6%	13.0%	7.1%	13.7%	14.1%

TABLE III: Performance of SHMEMGraph (second) when using Open MPI or Cray MPI.

	PR	CC	SSSP	BC	BFS
Open MPI	0.59	0.6	4.33	1.51	0.83
Cray MPI	0.51	0.54	4.04	1.39	0.75

TABLE IV: Comparison of SHMEMGraph and optimized single-thread implementation (second).

No. of thread	**1**	**2**	**4**
Single-thread	-	52.7	-
SHMEMGraph	109.1	54.3	26.1

C. Scalability

For scalability test, we firstly verify how SHMEMGraph can outperform the well-known single-thread optimized graph processing implementation [22]. Table IV shows the results. The single-thread implementation runs 20 iterations so their result is divided by 2. The table shows that SHMEMGraph outperforms the single-thread implementation at 4 threads. Since our work does not involve optimizations for computation efficiency or intra-node work balancing, the result is expected to be similar to the original Gemini [28]. This validates that SHMEMGraph's optimizations on communication and inter-node balancing are not contradictory to the existing strengths of the original frameworks that it builds upon.

Moreover, we evaluate the scale-out ability of SHMEMGraph. We increase the number of nodes being used for the same experiment. To have a better display, we normalize the job completion times to each algorithm 's single node performance. As shown in Fig. 8, SHMEMGraph can scale out nicely for most of the algorithms to 8 nodes. For computation-intensive algorithms such as PR and CC, the scaling to 8 nodes is better due to better distribution of computation costs. After 8 nodes, the performance scaling stalls or even worsens because now the inter-node communication occupies most of the total cost and both fine-grained data serving and communication balancing are not able to help with the performance. This indicates that the inefficiency instead of the imbalance becomes the main performance bottleneck for graph processing at scale and despite the use of RDMA, there is still room for more optimizations.

D. Design choices and Performance Tuning

To investigate the impact of the individual designs, we enable each component of SHMEMGraph individually. Fig. 11 shows the results for the three graphs with continuous vertices. Three cases of SHMEMGraph are evaluated: the basic one-sided communication channel (SHMEMGraph-basic), the fine-grained data serving on top of the basic channel (SHMEMGraph-FDS) and the communication balancing on top of the above two (SHMEMGraph-CB). From the figure, we can see that all components contribute to the performance to certain degrees. Firstly, the basic one-sided communication channel has the most consistent performance improvement compared to the original Gemini. Secondly, the fine-grained data serving leads to larger improvement in the tests that have a larger degree of computation imbalance (e.g. PageRank for enwiki-2013 and most of the algorithms in twitter-2010). Thirdly, the communication balancing delivers more consistent improvement than the fine-grained data serving. This is because the communication balancing not only well mitigates a larger degree of imbalance in some cases, but also reduces idle time for others by remote fetching active vertices.

1) Tuning Fine-grained Chunk Size: Fig. 9 shows the results tuning the fine-grained chunk size S_{chunk}. We plot Gemini's performance side-by-side for comparison. 256 MB is the largest chunking size for the testing graph (twitter-2010). We can see that the job completion time reaches the lowest point when the chunk size is 4 MB. After that, there are increasingly significant overheads from both the number of communication operations and thread coordination

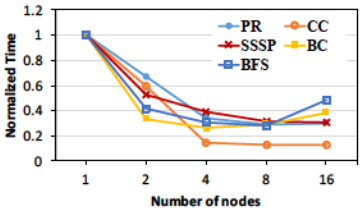


Fig. 8: Scalability.

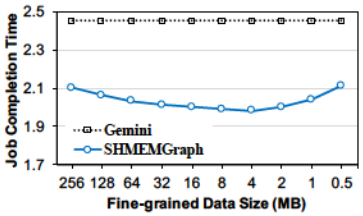


Fig. 9: Tuning S_{chunk}.

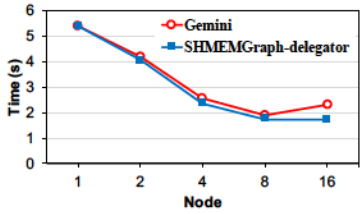
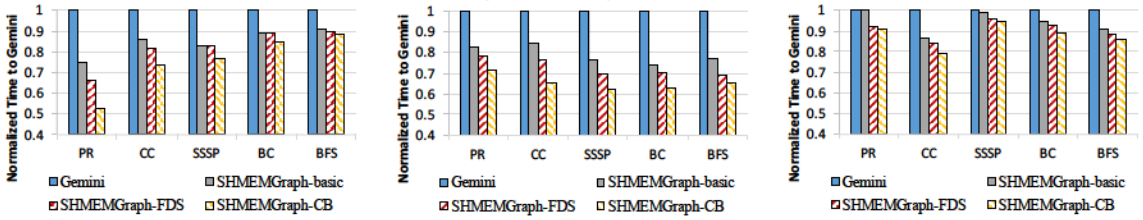


Fig. 10: Performance of delegator.

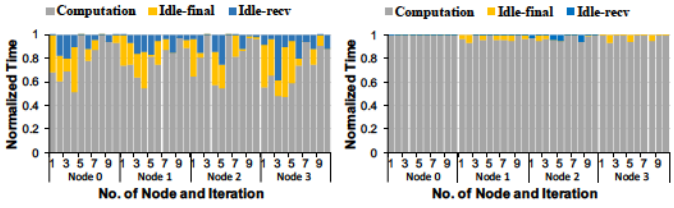


(a) enwiki-2013. (b) twitter-2010. (c) uk2007-05.

Fig. 11: Effect of SHMEMGraph designs.

on the receiving node. We do not evaluate network bandwidth here because the performance advantage of fine-grained data serving is mainly the reduced idle time on the receiver nodes, but not increased bandwidth. However, it will be interesting to see how this design can fit in with a different type of communication channel, e.g. a two-sided RPC, where we can further study the bandwidth behavior and comparison.

2) Delegator Performance: We also evaluate the performance of SHMEMGraph when using delegator instead of communication thread. As shown in Fig. 10, instead of similar performance improvement, SHMEMGraph with delegator has comparable or slightly better performance than that of Gemini. The main reason is the overheads added by memory copying between delegator and the processing engine. This negates the previous improvement that the original SHMEMGraph has over Gemini. However, for results on 16 nodes, Gemini has a considerable performance degradation due to lack of communication/computation overlapping when the communication work dominates the job completion time. In contrast, SHMEMGraph using delegator does not show such degradation because of its improved communication efficiency.



(a) Gemini. (b) SHMEMGraph.

Fig. 12: Mitigation of idle time.

E. Mitigation of Imbalance

To gain more insights into the ability of SHMEMGraph in mitigating the imbalance problem, we show the idle times of different nodes during the PageRank experiment. Fig. 12 shows the comparison between Gemini and SHMEMGraph. It is clearly shown that SHMEMGraph eliminates both two idle times to a large extent. With the same or even lower computation cost than Gemini, the elimination of idleness is directly reflected on the overall job performance improvement of SHMEMGraph as shown in previous sections.

Note that, the figure shows the time for each phase normalized to the total time spent on the iteration, so it is the longest idle portion in each iteration that shows the potential improvement we may have, but not the accumulative idle portions from all nodes. Also, the mitigation results indicate but not completely represent the improvement that SHMEMGraph has over Gemini. For example, the benefit from faster computation due to less threads being used is not reflected in them.

VI. Related Works

Leveraging one-sided communication for distributed data analytics frameworks, including graph processing algorithms and frameworks, has drawn increasing attention in recent years [14], [13], [9], [19], [26], [18]. Similar to our work, GraM [26] uses RDMA to accelerate communication for synchronous graph processing. However, its design for the RDMA-based RPC is closer to the delegator implementation of SHMEMGraph. Compared to SHMEMGraph's lightweight one-sided communication channel, GraM's design brought rather large overheads for both messaging buffer space and the number of sending/receiving threads. Moreover, Mizan-RMA [18] explores the use of MPI RMA operations for Mizan [15]. In contrast to aforementioned works, our basic one-sided communication channel not only leverages RDMA, but also further exploits the flexibility of one-sided operations by providing an optimal solution to the combined effect of imbalance. In addition, without carefully overlapping node-to-node communication/computation costs and reducing idle time, the aforementioned works still suffer from various imbalance issues discussed in this paper.

Many studies have pointed out the limitations of synchronous graph processing [20], [10], [24], [7]. For example, GraphLab [20] suggests using a more natural way to express asynchronous iterative computation. In general, synchronous model provides message batching, good convergence speed and computation/communication overlapping.

In contrast, asynchronous model eliminates synchronization barriers, allows both sync/async processing, but incurs lock contention and lacks message batching [27]. Unfortunately, the issue of node-to-node imbalance has been largely overlooked, despite that the issue is rooted in the essential limitation of synchronous processing as well. This is because this issue has been hidden well behind the imbalance in the overall iteration but became more obvious under finer-grained overlapping of inner-round communication/computation costs, like in Gemini.

Similar to our work, many studies attempt to balance the work between different nodes [10], [7], but they are more concerned about the imbalanced costs due to the power-law distribution of natural graphs. Others [25], [28], [3] try to reduce computation and communication needs by using a hybrid push/pull model. In contrast, based on an existing hybrid push/pull model of Gemini, our work moves beyond computation inefficiency and leverages one-sided `Put` and `Get` for more efficient and dynamic communication as well.

VII. Conclusion

We have examined the inefficiency and imbalance issues for state-of-the-art distributed and synchronous graph processing frameworks. We have proposed to build a graph processing framework on top of the global memory space and leveraged one-sided operations for its inter-node communication. We have implemented a prototype called SHMEMGraph based on Gemini and integrated a number of novel design ideas in order to address the inefficiency and imbalance. We have evaluated SHMEMGraph extensively on various graph data and graph algorithms. The results have demonstrated the performance advantages of SHMEMGraph over Gemini and also its ability to tackle both computation and communication imbalance.

Acknowledgment We are thankful to our shelp contact Dr. Martin Schulz and the anonymous shep/reviewers for their comments, and Amit Kumar Nath for his help on finalizing the paper. This work was supported in part by a contract from Oak Ridge National Laboratory and National Science Foundation awards 1561041, 1564647, and 1744336. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] Open MPI. https://www.open-mpi.org/.

[2] Titan Supercomputer. https://www.olcf.ornl.gov/titan/.

[3] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC17)*, 2017.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[7] 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*, page 1. ACM, 2015.

[8] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[9] H. Fu, M. G. Venkata, A. R. Choudhury, N. Imam, and W. Yu. High-performance key-value store on openshmem. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 559–568. IEEE Press, 2017.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[12] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[13] J. Jose, S. Potluri, H. Subramoni, X. Lu, K. Hamidouche, K. Schulz, H. Sundar, and D. K. Panda. Designing scalable out-of-core sorting with hybrid mpi+ pgas programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 7. ACM, 2014.

[14] J. Jose, S. Potluri, K. Tomko, and D. K. Panda. Designing scalable graph500 benchmark with hybrid mpi+ openshmem programming models. In *International Supercomputing Conference*, pages 109–124. Springer, 2013.

[15] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[16] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

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

[18] M. Li, X. Lu, K. Hamidouche, J. Zhang, and D. K. Panda. Mizan-rma: Accelerating mizan graph processing framework with mpi rma. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 42–51. IEEE, 2016.

[19] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda. Scalable graph500 design with mpi-3 rma. In *Cluster Computing, 2014 IEEE International Conference on*, pages 230–238. IEEE, 2014.

[20] 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 of the VLDB Endowment*, 5(8):716–727, 2012.

[21] 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 of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[22] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, 2015.

[23] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP'13*.

[24] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

[25] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[26] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[27] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[28] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.