BlitzG: Exploiting High-Bandwidth Networks for Fast Graph Processing

Yongli Cheng   Hong Jiang[‡]   Fang Wang   Yu Hua   Dan Feng

Wuhan National Lab for Optoelectronics, Wuhan, China

School of Computer, Huazhong University of Science and Technology, Wuhan, China

[‡]Department of Computer Science & Engineering, University of Texas at Arlington, USA

{chengyongli, wangfang, csyhua, dfeng}@hust.edu.cn   hong.jiang@uta.edu

Corresponding Author: Fang Wang

Abstract—Nowadays, high-bandwidth networks are easily accessible in data centers. However, existing distributed graph-processing frameworks fail to efficiently utilize the additional bandwidth capacity in these networks for higher performance, due to their inefficient computation and communication models, leading to very long waiting times experienced by users for the graph-computing results. The root cause lies in the fact that the computation and communication models of these frameworks generate, send and receive messages so slowly that only a small fraction of the available network bandwidth is utilized. In this paper, we propose a high-performance distributed graph-processing framework, called BlitzG, to address this problem. This framework fully exploits the available network bandwidth capacity for fast graph processing. Our approach aims at significant reduction in (i) the computation workload of each vertex for fast message generation by using a new slimmed-down vertex-centric computation model and (ii) the average message overhead for fast message delivery by designing a light-weight message-centric communication model. Evaluation on a 40Gbps Ethernet, driven by real-world graph datasets, shows that BlitzG outperforms the state-of-the-art distributed graph-processing frameworks by up to 27x with an average of 20.7x.

I. INTRODUCTION

Due to the wide variety of real-world problems that rely on processing large amounts of graph data [1]–[3], many vertex-centric distributed graph-processing frameworks, including Pregel [4], GraphLab [5], PowerGraph [6], GPS [7], Giraph [8] and Mizan [9], have been proposed to meet the compute needs of a wide array of popular graph algorithms in both academia and industry. These frameworks consider a graph-computing job as a series of iterations. In each iteration, vertex-associated work threads run in parallel across compute nodes. Common in these frameworks are a vertex-centric computation model and a vertex-target communication model [4], [7], as shown in Figure 1. In the *vertex-centric computation model*, the work threads on each compute node loop through their assigned vertices by using a user-defined **vertex-program(vertex *i*)** function. Each vertex program ingests the incoming messages (**Gather** stage), updates its status (**Apply** stage) and then generates outgoing messages for its neighbors (**Scatter** stage). The incoming messages were received from its neighboring vertices in the previous iteration. In the *vertex-target communication model*, the generated messages are first sent to the message buffers where the message batches are then sent to

the network. The message buffers are used to amortize the average communication overhead of each message [4], [7]. At the receiver side, the *message parser* receives the message batches and dispatches the messages in the message batches to the message queues of the destination vertices [4], [7]. Thus, the received messages can be identified by their destination vertices. The received messages serve as the inputs to their respective destination vertex programs in the next iteration.

A salient communication feature of vertex-centric distributed graph-processing frameworks is that, for most graph algorithms, the messages generated and delivered are usually small in size [10], [11]. Typically, a message carries a destination vertex name and a 4-byte integer or floating-point number. However, within each iteration, there are an enormously large number of messages used by vertices to interact with one another. This feature makes message delivery highly inefficient [8], [12], [13] and severely underutilizes the network bandwidth capacity even when the *message buffering* technique [4], [7] is used to amortize the average per-message overhead.

However, nowadays, high-bandwidth networks, such as 40Gbps, even 100Gbps networks, are easily accessible in data centers [14]. The large gap between the slow communication in existing distributed graph-processing frameworks and the easily accessible but severely underutilized high-bandwidth network capacity motivates us to fully exploit the high-bandwidth networks for high-performance graph computation. In order to fully exploit the high-bandwidth network capacity for fast graph computation, one must address the following two key challenges.

The first is that the messages must be generated fast enough. In order to generate a sufficient number of messages in a given time slot, one intuitive solution is to leverage expensive high-end servers with powerful processors to speed up the execution of vertex programs on each compute node [15], [16], because the messages are generated by the vertex programs. However, this solution may not be viable for most distributed graph-processing frameworks that are usually built on clusters of commodity computers with a limited number of cores, for better scalability and lower hardware costs [4]–[9]. Yet high scalability and low hardware cost are important considerations for graph-processing frameworks since a large number of compute nodes are required to process a large graph [4], [8].
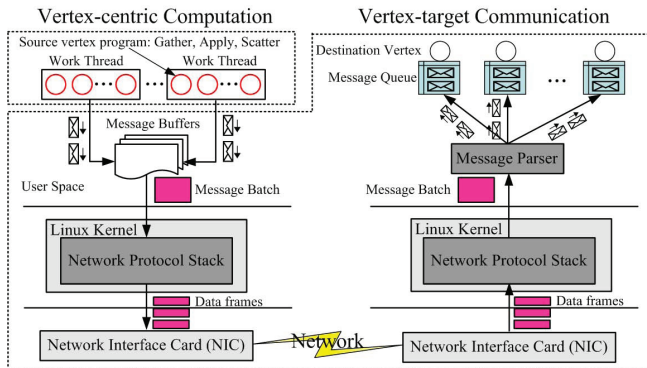
Fig. 1. A Pair of Compute Nodes in Vertex-Centric Distributed Graph-Processing Frameworks.

Instead of relying on more compute power, our proposed solution is to reduce the computation workload of each vertex. More specifically, since it is the vertex programs run by the work threads that generate the messages, we propose a slimmed-down vertex-centric computation model that helps eliminate the time-consuming **Gather** stage of the vertex program and thus significantly reduces the computation workload of each vertex. Our experimental results, as shown in Section VI, indicate that this method is very effective because the runtime of each vertex-program is dominated by the **Gather** and **Scatter** stages while the **Apply** stage entails a single simple operation of updating the value of the vertex [12].

The second challenge is that the average message time in existing distributed graph-processing frameworks is very long, which must be substantially shortened. That is, the messages among the vertices must be delivered fast enough. The average message time is defined as the time for sending an average message from a source vertex to a remote destination vertex. The long average message time is primarily consumed by the extra communication overheads of the kernel overhead, multi-copy overhead, interrupt overhead and the lock overhead [17]–[20], as discussed in Section II-B. We address this challenge by proposing a light-weight message-centric communication model that significantly reduces the average message time by avoiding the four extra communication overheads in existing distributed graph-processing frameworks, as discussed in Section IV.

Based on the proposed computation and communication models, we implement a high-performance distributed graph-processing framework, called BlitzG. This paper makes the following three contributions.

1) *A slimmed-down vertex-centric computation model* that significantly accelerates message generation by reducing the workload of each vertex.
2) *A light-weight message-centric communication model* that significantly reduces average message delivery time. Furthermore, this communication model significantly reduces the memory footprint by avoiding intermediate messages. Thus, our framework can support larger graphs or more complex graph algorithms with the same memory capacity, leading to lower hardware cost and better scalability.

3) *The design and prototype implementation of BlitzG* that can achieve the line-speed throughput of a 40Gbps Ethernet, for fast graph computation.

The rest of the paper is structured as follows. Background and motivation are presented in Section II. The proposed computation model is given in Section III. Section IV introduces the proposed communication model. Section V presents other key components of the BlitzG. Experimental evaluation of the BlitzG prototype is presented in Section VI. We discuss the related work in Section VII and conclude the paper in Section VIII.

II. BACKGROUND AND MOTIVATION

In this section, we first present a brief introduction to the *vertex-centric computation model* in existing distributed graph-processing frameworks. This helps motivate us to propose a new *slimmed-down vertex-centric computation model*, which can provide faster speed of message generation, as discussed in Section III. We then introduce the *vertex-target communication model*, in order to explore the high extra communication overheads of existing distributed graph-processing frameworks. The insights gained through these explorations help motivate us to propose a *light-weight message-centric communication model* that significantly reduces average message delivery time, as discussed in Section IV.

A. *Vertex-Centric Computation Model*

We discuss the execution process of a typical compute node in the vertex-centric distributed graph-processing frameworks to help understand the vertex-centric computation model. In this model, the graph to be processed is first partitioned by a predefined scheme so that each subgraph is loaded to a compute node that then assigns its vertices to a limited number of work threads, each of which loops through its assigned vertices by using a user-defined **vertex-program(vertex i)**. As shown in Figure 2(a), **vertex-program(vertex i)** sequentially executes the following three stages for each vertex i: Gather, Apply, and Scatter. In the **Gather** stage, the value of each message in the input message queue MQ_input[i] of vertex i is collected through a generalized sum:

$$\sum \leftarrow \bigoplus_{m \in MQ\_input[i]} m.value \qquad (2.1)$$

The user defined sum $\bigoplus$ operation is commutative and associative, and can range from a numerical sum to the union of the data on the incoming messages [5].

The resulting value Σ is used in the **Apply** stage to update the value of the vertex i (indicated as **Y**):

$$f(Y, \Sigma) \rightarrow Y \qquad (2.2)$$

Finally, in the **Scatter** stage, vertex i uses its new value (**Y**) to generate messages and then sends these messages to its outgoing neighbors. Usually, in order to improve communication efficiency, the messages are batched in the dedicated message buffers [4], [7] before sending them over the network.

(a) vertex-program(vertex i).
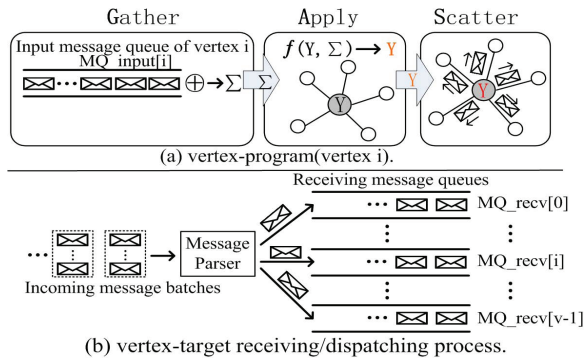
(b) vertex-target receiving/dispatching process.

Fig. 2. The Workflow of the Vertex-Centric Distributed Graph-Processing Framework in A Typical Compute Node.

Once the vertex-centric computation model is activated, the compute node begins to execute the vertex-target receiving/dispatching process concurrently. This process is an integral part of the vertex-target communication model, as discussed in the next subsection. As shown in Figure 2(b), each incoming message batch is received by a *message parser* [7]. The *message parser* thread parses each message batch and enqueues the messages in the message batch into the message queues of the destination vertices. Thus, each vertex can identify the messages sent to itself.

Each vertex *i* has two message queues, i.e., the receiving message queue MQ_recv[*i*] and the input message queue MQ_input[*i*] [4], [7]. The former is used to store the messages that are sent to vertex *i*. The latter stores the messages received in the previous iteration and serves as the input to the **vertex-program**(**vertex *i***). At the end of each iteration, the two message queues switch their roles.

B. Vertex-Target Communication Model

As shown in Figure 1, the vertex-target communication model works as follows. At the sender side, any message generated by a work thread is first sent to the user-space message buffers [4], [7]. When a message buffer is filled up, the message batch is delivered to kernel network protocol stack where it is sent to the network. At the receiver side, when a message batch is received by the kernel network protocol stack, it is first delivered to the user-space. The *message parser* then parses the message batch and enqueues the messages in the message batch to the message queues of the destination vertices [4], [7]. As mentioned before, this communication model usually suffers from four extra communication overheads, leading to long average message delivery time.

Kernel overhead. Existing distributed graph-processing frameworks are built on an operating system kernel communication protocol stack where the message batches are passed through the network [4], [7]. Modern operating system kernels provide a wide range of general networking functionalities. This generality does, however, come at a performance cost, severely limiting the packet processing speed [17].

Multi-copy overhead. In high-bandwidth networks, excessive data copying results in poor performance [18], [21].

(a) slimmed-down vertex-program(*vertex i*).

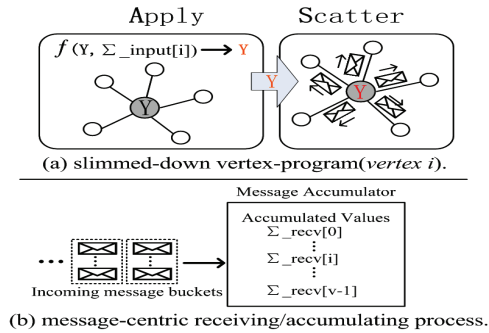(b) message-centric receiving/accumulating process.

Fig. 3. The Workflow on A Typical Compute Node of BlitzG.

However, data copying occurs twice each at the sender side and at the receiver side in existing vertex-centric distributed graph-processing frameworks, as shown in Figure 1.

Interrupt overhead. Conventional network interface card (NIC) drivers usually use interrupts to notify the operating system that data is ready for processing. However, interrupt handling can be expensive in modern processors, limiting the packet receiving speed [19], [20], [22], [23].

Lock overhead. Contention among threads on critical resources via locks is a potential bottleneck that prevents the high-bandwidth network capacity from being efficiently utilized by distributed graph-processing frameworks [20].

III. SLIMMED-DOWN VERTEX-CENTRIC COMPUTATION MODEL

We present the execution process of a typical compute node in our BlitzG framework to help understand our slimmed-down vertex-centric computation model. In this model, like the existing vertex-centric computation model, each work thread in the compute node loops through its vertices by using the **vertex-program(vertex *i*)**. Unlike existing vertex-centric computation model, as shown in Figure 3(a), the **vertex-program(vertex *i*)** sequentially executes the Apply and Scatter stages only, for each vertex *i*. In the **Apply** stage, the input accumulated value $\Sigma\_input[i]$ is used to update the value of the vertex *i* (indicated as **Y**) :

$$f(Y, \Sigma\_input[i]) \rightarrow Y \qquad (3.1)$$

In the **Scatter** stage, vertex *i* uses its new value (**Y**) to generate messages for its outgoing neighbors. The messages are constructed directly in the *message buckets*. A *message bucket* is a message container consisting of the Ethernet header, IP header and a number of data structures of messages, as discussed in Section IV. When a message bucket is full, it is sent to the network interface card (NIC) directly.

Once the slimmed-down vertex-centric computation model is activated, the compute node begins to execute the message-centric receiving/accumulating process concurrently. This process is an integral part of the light-weight message-centric communication model, as detailed in Section IV. As shown in Figure 3(b), instead of the *message parser*, a *message accumulator* is used to ingest the incoming *message buckets*

where the value of each message *msg* in the message buckets is accumulated through a generalized sum:

$$\Sigma\_recv[\boldsymbol{i}] \oplus msg.value \rightarrow \Sigma\_recv[\boldsymbol{i}] \qquad (3.2)$$

where $\boldsymbol{i}$ is the destination vertex name of the message *msg*, and $\Sigma\_recv[\boldsymbol{i}]$ is the receiving accumulated value of vertex $\boldsymbol{i}$.

Each vertex $\boldsymbol{i}$ is associated with two user-defined accumulated values, i.e., a receiving accumulated value $\Sigma\_recv[\boldsymbol{i}]$ and an input message accumulated value $\Sigma\_input[\boldsymbol{i}]$. The former is used to accumulate the values of the incoming messages that are sent to vertex $\boldsymbol{i}$. The latter serves as the input of the **vertex-program**(**vertex** $\boldsymbol{i}$). At the end of each iteration, the two accumulated values switch their roles.

Summary: Our slimmed-down vertex-centric computation model speeds up the message generation by eliminating time-consuming **Gather** stage of the vertex program. This gain is attributed to the message-centric receiving/accumulating process. Like vertex-target receiving/dispatching process in the vertex-centric distributed graph-processing frameworks, our message-centric receiving/accumulating process needs to parse the received message buckets to identify each message, resulting in parsing overhead [7]. However, instead of dispatching the identified messages to their destination vertices, our approach accumulates the values of the identified messages to the *accumulators* of the destination vertices directly, avoiding the overheads of message migrations. In the next iteration, the value of each *accumulator* is used by the **Apply** stage of the destination vertex program directly. More importantly, our approach avoids the costly kernel, multi-copy, interrupt and defragmenting overheads, enabling the messages to be received/processed efficiently, as discussed in Section IV.

IV. LIGHT-WEIGHT MESSAGE-CENTRIC COMMUNICATION MODEL

Our light-weight message-centric communication model, as shown in Figure 4(c), is able to significantly reduce the communication time of an average message primarily because of the following reasons.

First, in order to avoid kernel overhead of operating system, our communication model first employs the Intel Data Plane Development Kit (DPDK) [19], a fast user-space packet processing framework that has been gaining increasing attention [14], [19], [20], [23]. DPDK allows user-space applications using the provided drivers and libraries to access the Ethernet controllers directly without needing to go through the Linux kernel. These libraries can be used to receive and send packets within a minimum number of CPU cycles, usually less than 80 cycles, in contrast to the approximately 200 cycles required to access the memory [19]. To the best of our knowledge, this is the first time that DPDK is used in graph-processing frameworks. The high performance of DPDK make it possible for BlitzG to utilize the high-bandwidth networks efficiently.

Second, BlitzG eliminates the four rounds of data copying in vertex-centric distributed graph-processing frameworks. This is important since, in high-bandwidth networks, excessive data copying results in poor performance [18], [21]. At the sender side, when each work thread executes its vertex programs, it updates the *DestinationVertexID* and *MessageValue* fields of each message in the message buckets directly, avoiding the message migrations to the message buffers in existing vertex-centric distributed graph-processing frameworks. As shown in Figure 4(a), a *message bucket* contains the following fields: the Ethernet header, IP header, *BuckNum*, *ACKNum*, a number of messages (payload) and pad. The *BucketNum* and *ACKNum* fields are used to guarantee reliable transmission. We employ the DPDK *mbuf* data structure [19] to store the *message bucket* that is indexed by a pointer. When a message bucket is full, it is flushed to the network interface card (NIC) directly by using the user-space DPDK drivers deployed by the *message bucket sender*, avoiding the data copying from the use space to the kernel space. At the receiver side, the *message accumulator* receives the message buckets from the NIC and accumulates the message value of each message in the message buckets directly to the $\Sigma\_$**recv[$\boldsymbol{i}$]** variable, where $\boldsymbol{i}$ is the vertex name appearing in the message, avoiding the data copying from the kernel space to the user space and the message migrations from the use-space buffers to the message queues of the destination vertices. The accumulated values serve as the inputs to the destination vertex programs in the next iteration.

Third, instead of using interrupts to signal packet arrival, the receive queues of network controllers are polled by the receiving threads directly, avoiding the costly interrupt overheads.

Fourth, our communication model avoids the extra overheads of packet fragmenting/defragmenting. Existing vertex-centric distributed graph-processing frameworks usually use large-size message buffers to amortize the average overhead of each message [4], [7]. In this case, fragmenting/defragmenting is required so that each message batch can be encapsulated within data frames that have a size constrained by the Maximum Transmission Unit (MTU). For example, the MTU of Ethernet network is typically 1500 Bytes. However, packet fragmenting/defragmenting can decrease the efficiency of packet processing when large-size packets pass through the networks [24]. Instead of large size, our communication model avoids the fragmenting/defragmenting by limiting the message bucket size to be slightly smaller than the MTU. This design is based on the fact that the messages of most graph algorithms are usually short and have a uniform size [10], [11]. Our communication model also supports jumbo messages, each of which is composed of multiple message buckets linked together through their "next" field, albeit a rare case scenario.

Finally, the high performance of our communication model also stems from the lockless design, as discussed in Section V-A.

Summary: The proposed light-weight message-centric communication model significantly reduces the communication time of average message by avoiding the costly extra communication overheads in existing vertex-centric distributed graph-processing frameworks.

Furthermore, this communication model is highly memory-saving because it ingests the values of incoming messages
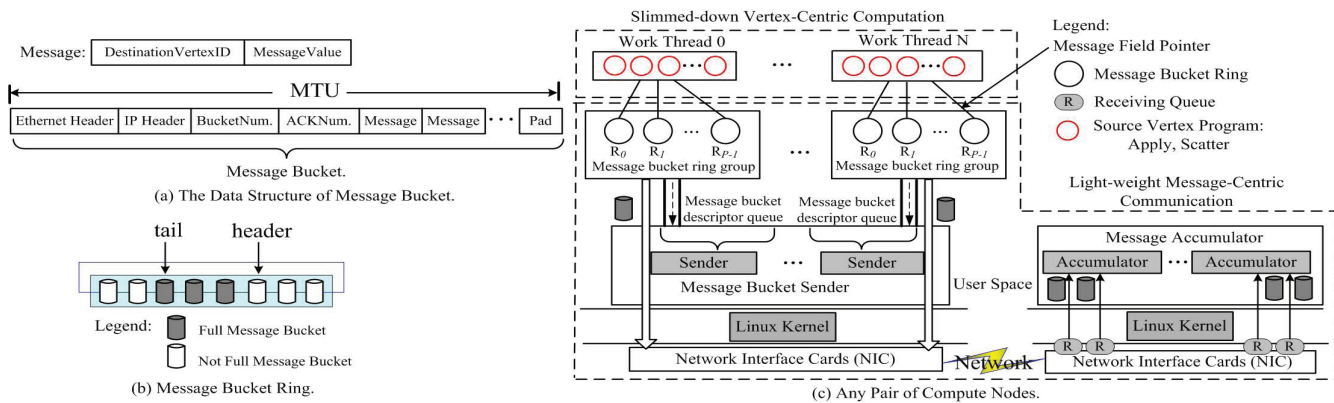
Fig. 4. The BlitzG Framework.

directly on the fly, eliminating intermediate messages in vertex-centric distributed graph-processing frameworks. Memory consumption is an important concern in graph-processing systems [25]. Because, given the aggregate memory capacity of the compute nodes in a cluster, the memory-saving graph-processing systems are able to process larger graphs or more complex graph algorithms, leading to lower hardware cost.

V. DESIGN & IMPLEMENTATION OF BLITZG

BlitzG is able to achieve the line-rate throughput of high-bandwidth networks due to the slimmed-down vertex-centric computation model and the light-weight message-centric communication model, as discussed in Sections III and IV. In this section, we focus mainly on other key components of BlitzG.

A. Lockless Design

Intuitively, the work threads in each compute node can share the message buckets. Shared memory is typically managed with locks for data consistency, but locks inevitably degrade performance by serializing data accesses and increasing contention [20], [26]. To address this problem, we propose parallelized *message bucket ring groups*, each of which serves for one work thread. As shown in Figure 4(b), a *message bucket ring* consists of a set of *message buckets* that are used to store messages with the same remote compute node as their destination. As shown in Figure 4(c), each work thread has a *message bucket ring group* that includes P-1 *message bucket rings*, where P is the number of compute nodes. Each *message bucket ring* is dedicated to one remote compute node independently. The message fields of each *message bucket ring* are updated by its work thread sequentially in order by using the automatically incremented value of a message field pointer. When all the message fields in a message bucket are updated, the message bucket is marked to be full, and its descriptor is sent to the message *bucket descriptor queue* of the *message bucket ring group*.

The *message bucket sender* module has a number of *sender* threads, each of which manages several *message bucket descriptor queues*, as shown in Figure 4(c). Each *sender* polls its *message bucket descriptor queues*. When a message bucket descriptor is obtained from a *message bucket descriptor queue*, the *message bucket* indexed by the descriptor is directly sent

to the NIC. Using this design, each *sender* can also work independently without accessing any shared data.

Next we discuss the lockless design of message bucket receiving/processing. Modern NICs are usually supported by the Receiver-Side Scaling (RSS) technique [14] with multiple queues that allow the packet receiving and processing to be load balanced across multiple processors or cores [27]. For example, the Mellanox ConnectX-3 NIC has up to 32 queues [14]. In order to completely parallelize the message bucket receiving/processing, the *message accumulator* module has multiple accumulator threads, each of which manages several receiving queues of the NIC. By using this design, each accumulator thread is allowed to work independently.

Summary: There are three key components in our BlitzG framework, i.e., computation(work threads), *message bucket sender* and *message accumulator*. Due to the lockless design, the work of each key component is parallelized by multiple dedicated threads, each of which works independently. Using this design, BlitzG obtains high scalability in terms of core count that enables high-bandwidth network to be fully utilized.

Furthermore, due to the lockless design, each thread of the three key components is busy all the time, enabling the dedicated cores to be utilized efficiently. Once the work threads begin to work, the accumulator threads receive and then process the message buckets continuously, significantly reducing the polling time for the arrivals of the message buckets.

B. Reliable Transmission

Reliable transmission must be ensured even though the dropped packet rate is very low in a high-quality network. BlitzG only needs to guarantee reliable transmission between any pair of compute nodes.

"Sending with Acknowledgement" Mechanism : At the beginning of each iteration, each side of a pair of compute nodes begins to send message buckets to its peer. The sent message buckets are numbered sequentially. The sequence number of each message bucket is carried in the *BucketNum* field in the message bucket. The *AckNum* field in each message bucket is used to inform the peer that all the message buckets with their sequence number being less than or equal to the *AckNum* value have been received. When the expected *AckNum*

value has not been received within an expected time interval, the message buckets with their sequence number being larger than the last received *AckNum* value will be re-sent to the peer. Message bucket transmission between any two compute nodes proceeds with this "sending with acknowledgement" mechanism. When a compute node sends the last message bucket to its peer, a "last" flag is carried in the *BucketNum* field in the message bucket to inform the peer to stop receiving. In this case, the compute node without message buckets to send is still ready for receiving message buckets from its peer until the "last" signal is received.

Delayed Processing Policy: In our "sending with acknowledgement" mechanism, a *connection descriptor* is designed for any given pair of compute nodes. These *connection descriptors* are used by the *sender* threads and the *accumulator* threads to trace and guarantee the reliable message transmission process between any pair of compute nodes. Collisions can occur in some cases. For example, when two different accumulator threads on the same compute node have received their respective message buckets from the same remote compute node, each of them needs to process its message bucket and update the status of the same *connection descriptor* simultaneously. In this case, in order to avoid contention overheads to achieve higher performance, our approach is to delay the processing of any message bucket that has a *BucketNum* being larger than a predefined threshold. The delayed thread continues to receive the subsequent message buckets, and the delayed message buckets will be processed, along with the new ones in the correct order. This approach enables each communication thread to work independently, improving the utilization of cores and system scalability.

Algorithm 1: Collect(*msg*) Function of PageRank

int i;
i = msg.DestinationVetexID;
$\sum \_recv[i]$+=msg.value;

**Algorithm 2: Vertex-Program(vertex *i*) Function of PageRank**

vertices[i].value = $\sum \_input[i]$*0.85+0.15; /* Apply */
foreach(j in out_neighbors of vertex i) /* Scatter */
    SendMsg(j, vertices[i].value);

C. Programming API

To better illustrate the programming API, we use an example of the computation of PageRank [28]. In each iteration, each compute node has two independent processes. In the message-centric receiving/accumulating process, the *message accumulator* ingests the incoming *message buckets*. Each message *msg* in the *message buckets* is processed by using the **collect(*msg*)** function, as shown in **Algorithm 1**. In the slimmed-down vertex-centric computation model, each work thread loops through its assigned vertices by using the **vertex-program(vertex *i*)** function of PageRank. This function first executes the Apply stage then the Scatter stage, as shown in **Algorithm 2**.
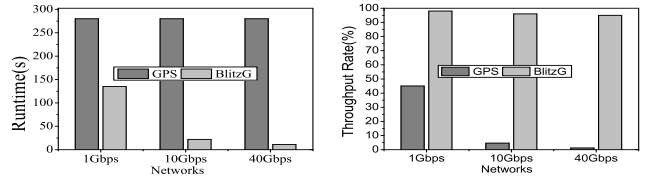
(a) Runtime.　　　　　(b) Throughput Rate.

Fig. 5. Performance Analysis. The "throughput rate" is defined as the throughput value normalized to the line-speed throughput of the network, i.e., throughput rate = throughput /(line-speed throughput of the network).
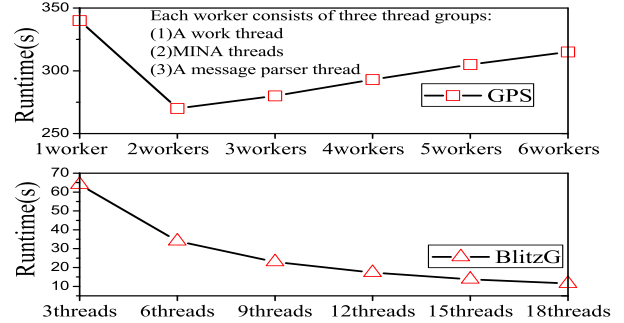

Fig. 6. Impact of Number of Threads.

VI. EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to evaluate the performance of BlitzG. Experiments are conducted on a 32-node cluster. Each compute node has two 6-core Intel(R) Xeon(R) E5-2620 processors with 32GB of RAM and a Mellanox ConnectX-3 VPI NIC. Each core has two logical cores by means of the hyper-threading technology. Nodes are connected via a 40Gbps Ethernet network. We use 8GB of RAM for huge pages [19], based on the fact that in Intel's performance reports 8GB is set as a default huge page size. The operating system of each node is CENTOS 7.0 (kernel3.10.0). DPDK-2.1_1.1 is used.

Graph Algorithms and Datasets: We evaluate BlitzG by implementing several graph algorithms, such as Single-Source Shortest-Paths (SSSP) [31], PageRank (PR) [28], Community Detection (CD) and Connected Components (CC) [12]. We evaluate BlitzG using several real-world graph datasets that are summarized in Table I.

Baseline Framework: We compare BlitzG with GPS, which is an open-source Pregel implementation from Stanfords InfoLab [7]. It is a representative vertex-centric distributed graph-processing framework.

A. Performance Analysis

BlitzG is compared with GPS in terms of the total runtime and throughput. Each framework, built on a 24-node cluster, runs the PageRank algorithm with the Twitter-2010 on the 1Gbps, 10Gbps and 40Gbps Ethernets respectively. GPS is evaluated with its default size of message buffers [7].

The experimental results shown in Figure 5(a) indicate that the runtime of GPS is not improved on a higher-bandwidth network due to two factors, i.e., the slow process of message generation and the costly extra communication overheads. The two factors prevent GPS from utilizing the network bandwidth

TABLE I
GRAPH DATASETS SUMMARY.

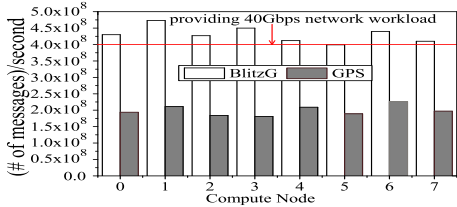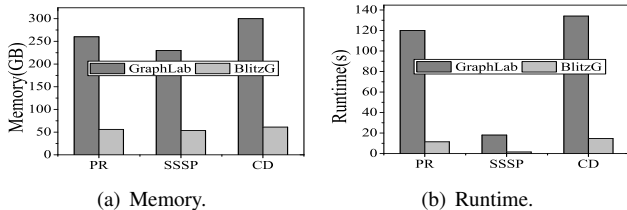| DataSets	$|V|$	$|E|$	Type	Avg/In/Out degree	Max -/In/Out degree	Largest SCC
LiveJournal [29]	4.8×10^6	69×10^6	Social Network	18/14/14	20K/13.9K/20K	3.8M (79%)
Twitter-2010 [30]	41×10^6	1.4×10^9	Social Network	58/35/35	2.9M/770K/2.9M	33.4M (80.3%)
UK-2007-05 [30]	106×10^6	3.7×10^9	Web	63/35/35	975K/15K/975K	68.5M (64.7%)

Fig. 7. Speed of Message Generation.

(a) Memory.   (b) Runtime.

Fig. 8. Memory Consumption & Performance.

capacity effectively, achieving only 45%, 4.6% and 1.2% of line-speed throughputs respectively of the 1Gbps, 10Gbps and 40Gbps Ethernet networks, as shown in Figure 5(b).

However, BlitzG obtains significant performance improvement on a higher-bandwidth network. The reasons are twofold. First, the slimmed-down vertex-centric computation model that significantly accelerates message generation by reducing the workload of each vertex. Second, the light-weight message-centric communication significantly reduces the communication time of average message by eliminating the costly extra communication overheads. The faster message generation and the shorter communication time of average message enable the network bandwidth capacities to be utilized at 98%, 96% and 95% respectively when running on the 1Gbps, 10Gbps and 40Gbps Ethernet networks. In these experiments, BlitzG is 2.1x, 12.7x and 25.3x faster than GPS respectively when running on the 1Gbps, 10Gbps and 40Gbps Ethernet networks. These experimental results indicate that BlitzG significantly outperforms GPS in terms of runtime, especially when a higher-bandwidth network is available.

B. *Impact of Core Count*

We compare BlitzG with GPS in terms of the impact of the number of cores in each compute node. Each framework with a 24-node cluster runs 10 iterations of PageRank with the Twitter-2010. GPS parallelizes a graph-computing job by using multiple workers in each compute node [7]. Each worker has a work thread, a message parsing thread and several MINA (an Apache network application framework) threads [7], [32]. To efficiently utilize the CPU cores, MINA sets the size of the thread pool as "number of logical cores + 1" by default. Hence, we study GPS in terms of the impact of the number

of workers in each compute node. GPS runs repeatedly by increasing the number of workers in each compute node. Experimental results, as shown in Figure 6, indicate that GPS has a sweet spot at 2 workers per compute node: adding more workers degrades performance. The reason is contention for the shared message buffers [33]. In each experiment, the number of threads used by GPS in each compute node is larger than 24, the number of logical cores in each compute node. Most of the dedicated threads are used by the MINA. This indicates that the communication load of GPS is heavy.

Unlike GPS, BlitzG uses multithreading in its three key modules of each compute node: computation (work threads), message bucket sender and message bucket accumulator. Instead of using a thread pool, BlitzG binds each thread to a dedicated logical core. BlitzG runs repeatedly by increasing the number of threads of each compute node, that is, each compute node is assigned 3, 6, 9, 12, 15 and 18 threads in different experiments, as shown Figure 6. Experimental results show that the runtime of BlitzG is reduced gradually until reaching the peak performance when 18 threads are used by each compute node. We also conduct experiments to identify the minimum numbers of required threads for its three key modules (i.e., computation, message bucket sender and message bucket accumulator) that contribute to the peak performance of BlitzG. Experimental results indicate that BlitzG can reach its peak performance by assigning 4 threads to message bucket sender, 6 threads to message bucket accumulator, and 6 threads to computation. The reason is that the message bucket sender has lighter workload.

C. *Speed of Message Generation*

In order to study the effectiveness of our slimmed-down vertex-centric computation model, we compare BlitzG with GPS in terms of message generation speed. Each framework is run on an 8-node cluster executing the PageRank algorithm with the Twitter-2010. In each compute node, BlitzG assigns 4 threads to the message bucket sender, 6 threads to the message bucket accumulator, and 6 threads to computation since the experimental results, as shown in Section VI-B, indicate that BlitzG with this configuration is able to achieve near-line-speed throughput of the 40Gbps network. For fair comparison, GPS assigns 4 workers to each compute node, with each worker consisting of one work thread, one message parser thread and 2 MINA threads, for a total of 16 threads. Both the size of the message buffers in GPS and the size of the message bucket rings in BlitzG are set at a sufficiently large value respectively to avoid idle times experienced by work threads to wait for the communication.
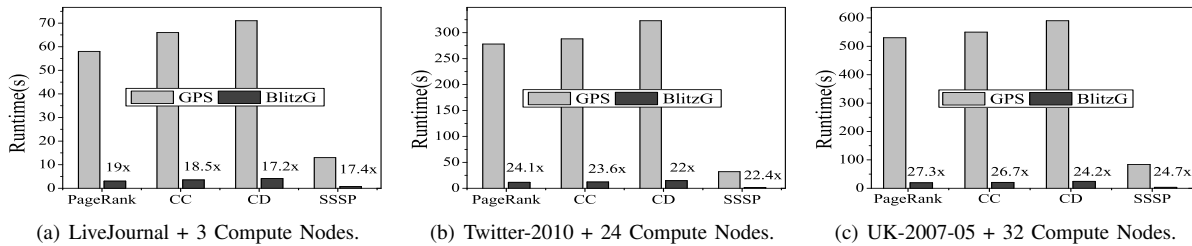
(a) LiveJournal + 3 Compute Nodes.   (b) Twitter-2010 + 24 Compute Nodes.   (c) UK-2007-05 + 32 Compute Nodes.

Fig. 9. Comprehensive Evaluation on Different Graphs & Graph Algorithms.

In each experiment, the message generation speed of each compute node is calculated according to the number of total generated messages of the compute node and the runtime of the slowest work thread in the compute node. Experimental results, as shown in Figure 7, indicate that the message generation speeds of compute nodes in BlitzG range from 4.02×10^8 to 4.73×10^8 messages/second. In the PageRank algorithm, each message consists of an 8-byte long-integer number to carry the destination vertex name and a 4-byte floating-point number to carry the pagerank value. In this case, the message generation speed of each compute node in BlitzG can provide sufficient communication workload to fully utilize the 40Gbps network if the communication model is also efficient enough. However, each compute node in GPS fails to provide so fast message generation speed as BlitzG. The message generation speeds of the compute nodes in GPS are only 43%-51% of those of BiltzG. The message generation speeds are sufficient for GPS due to its slower communication model, but insufficient for BlitzG due to its highly efficient communication model. Intuitively, GPS can also provide fast message generation speed as BlitzG by using more work threads. However, as mentioned in Section I, most existing distributed graph-processing frameworks are built on commodity compute nodes with limited core count for low hardware costs and better scalability. Furthermore, as verified in Section VI-B, the communication workload of GPS is heavier than its computation workload. In this case, increasing the number of work threads can deprive CPU resources of communication threads, leading to worse overall system performance.

D. Memory Consumption & Performance

MOCgraph [25] can achieve a similar performance to GraphLab [5] with significantly smaller memory consumptions. GraphLab is an open-source project originated at CMU [5] and now supported by GraphLab Inc. It is a representative distributed shared-memory graph-processing framework. We also compare BlitzG with GraphLab in terms of memory consumption and performance. We use the latest version of GraphLab 2.2, which supports distributed computation and incorporates the features and improvements of PowerGraph [5], [6]. Each framework with a 24-node cluster runs SSSP, PR and CD respectively on the Twitter-2010. Figure 8 shows the experimental results. Although the memory consumption of GraphLab is about 4.3x-5.6x larger than that of BlitzG, BlitzG can achieve 9.6x-11.8x performance improvement over

GraphLab. The reasons are twofold. First, like MOCgraph, BlitzG greatly reduces the memory footprint by significantly reducing intermediate data. Second, unlike MOCgraph, BlitzG significantly speeds up the message generation and reduces the communication time of average message, leading to the higher performance.

E. Comprehensive Evaluation

We evaluate BlitzG comprehensively with different graph algorithms on various graph datasets against GPS. Each framework runs four graph algorithms on various graph datasets (as shown in Table I). We run PageRank with 10 supersteps on each graph dataset. Experiments are conducted on three clusters with 3 compute nodes, 24 compute nodes and 32 compute nodes respectively when using the LiveJournal, Twitter-2010 and UK-2007-05 graphs as inputs. The experimental results are shown in Figure 9. We use the experimental results of GPS as the baselines for an easy two-way comparison.

Experimental results indicate that the speedup of BlitzG is higher when running on bigger graph datasets. For example, BlitzG is 19x, 24.1x and 27.3x faster than GPS respectively when running PR on the LiveJournal, Twitter-2010 and UK-2007-05 graphs. BlitzG's higher scalability on larger graph datasets results from the fact that more compute nodes are required by bigger graph dataset, which in turn leads to higher speedup. Furthermore, BlitzG is demonstrated to significantly outperform GPS across a range of different algorithms consistently with similar level of improvement in performance. In our comprehensive evaluation, BlitzG runs 17.2x-27.3x (with an average of 20.7x) faster than GPS respectively on various graph algorithms and graph datasets.

VII. RELATED WORK

In this section, we briefly discuss the work on distributed graph-processing frameworks most relevant to our BlitzG.

The message buffers technique is used by most distributed graph-processing frameworks to amortize the average overhead of each message [4], [7]–[9]. This technique can improve the communication efficiency by sending the message batches. However, these frameworks still suffer from long communication time of average message due to the costly kernel, multi-copy, interrupt and fragmenting/defragmenting overheads. BlitzG avoids these costly extra overheads by using the light-weight message-centric communication model.

Sedge [34] aims to reduce the inter-node communication by graph partitioning. Mizan [9] introduces dynamic load balancing and efficient vertex migration to reduce the number

of messages. GPS [7] introduces the dynamic repartitioning and large adjacency list partitioning (LALP) techniques to reduce the number of messages sent over the network. Pregel [4] adopts a combiner to reduce the number of cross-machine messages. These methods reduce the number of inter-node messages, lowering the communication costs, which are also effective in our BlitzG.

MOCgraph [25] reduces the memory footprint by significantly reducing intermediate data. This approach is very useful for processing larger graphs or more complex graph algorithms within the same memory capacity. Like MOCgraph, BlitzG is memory-saving due to its light-weight message-centric communication model. Unlike MOCgraph, our communication model aims to reduce the communication time of average message by avoiding the costly extra communication overheads, as mentioned before.

The DPDK framework is proposed recently to provide capacities of fast packet processing in software [19], which has been gaining increasing attention. Using this framework, NetVM [20] brings virtualization to the network by enabling high bandwidth network functions to operate at near line speed. While BlitzG employs the DPDK technology to improve the communication efficiency in distributed graph-processing frameworks.

VIII. **CONCLUSION**

In this paper, we propose a distributed graph-processing framework, called BlitzG. This framework fully exploits the modern high-bandwidth networks for fast graph computation by significantly speeding up the message generation and reducing the communication time of average message. Extensive prototype evaluation of BlitzG, driven by real-world datasets, indicates that it can achieve nearly line-speed throughout of a 40Gbps Ethernet, gaining 17.2x-27.3x (with an average of 20.7x) performance improvement over state-of-the-art distributed graph-processing frameworks.

ACKNOWLEDGMENT

REFERENCES

[1] T. Friedrich and A. Krohmer, "Cliques in hyperbolic random graphs," in *Proc. IEEE INFOCOM'15*.
[2] X. Wang, R. T. Ma, Y. Xu, and Z. Li, "Sampling online social networks via heterogeneous statistics," in *Proc. IEEE INFOCOM'15*.
[3] X.-Y. Li, C. Zhang, T. Jung, J. Qian, and L. Chen, "Graph-based privacy-preserving data publication," in *Proc. IEEE INFOCOM'16*.
[4] G. Malewicz, M. H. Austern, and etc., "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD'10*.
[5] Y. Low, D. Bickson, and etc., "Distributed graphlab: a framework for machine learning and data mining in the cloud," in *Proc. VLDB'12*.
[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," *Presented as part of the 10th USENIX Symposium on OSDI'12*.
[7] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proc. ACM SSDBM'13*.
[8] "Apache giraph." http://giraph.apache.org.
[9] 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 *Proc. ACM EuroSys'13*.
[10] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.
[11] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.
[12] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc." *Presented as part of the 10th USENIX Symposium on OSDI'12*.
[13] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, and X. Wang, "Ddgraph: A highly cost-effective distributed disk-based graph-processing framework," in *Proc. ACM HPDC'16*.
[14] "Mellanox," http://www.mellanox.com/.
[15] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proc. SC'14*.
[16] 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 ACM Symposium on Cloud Computing*, 2015.
[17] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*. IEEE Computer Society, 2015, pp. 5–16.
[18] P. Druschel and G. Banga, "Lazy receiver processing (lrp): A network subsystem architecture for server systems," in *OSDI*, vol. 96, 1996, pp. 261–275.
[19] "Dpdk." http://www.dpdk.org.
[20] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 34–47, 2015.
[21] L. Rizzo, "netmap: A novel framework for fast packet i/o." in *USENIX Annual Technical Conference*, 2012, pp. 101–112.
[22] C. Dovrolis, B. Thayer, and P. Ramanathan, "Hip: hybrid interrupt-polling for the network interface," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, 2001.
[23] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *Proc. FAST'12*.
[24] S. A. Athalye and T. Ji, "Method and apparatus for fragmenting a control message in wireless communication system," 2011.
[25] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *Proc. VLDB'14*.
[26] Y. Wu, F. Wang, Y. Hua, D. Feng, Y. Hu, J. Liu, and W. Tong, "Fast fcoe: An efficient and scale-up multi-core framework for fcoe-based san storage systems," in *Proc. IEEE ICPP'15*.
[27] S. Makineni, R. Iyer, P. Sarangam, D. Newell, L. Zhao, R. Illikkal, and J. Moses, "Receive side coalescing for accelerating tcp/ip processing," in *High Performance Computing-HiPC 2006*. Springer, pp. 289–300.
[28] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.
[29] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proc. ACM SIGKDD'06*.
[30] "The laboratory for web algorithmics." http://law.di.unimi.it/datasets.php.
[31] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par 2014 Parallel Processing*. Springer, 2014, pp. 451–462.
[32] "Mina," http://mina.apache.org/.
[33] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB'14*.
[34] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proc. ACM HPDC'14*.