This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

Using High-Bandwidth Networks Efficiently for Fast Graph Computation

Yongli Cheng, *Member, IEEE,* Hong Jiang, *Fellow, IEEE,* Fang Wang, *Member, IEEE,* Yu Hua, *Senior Member, IEEE,* Dan Feng, *Member, IEEE,* Wenzhong Guo and Yunxiang Wu

Abstract—Nowadays, high-bandwidth networks are more easily accessible than ever before. However, existing distributed graph-processing frameworks, such as GPS, 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 GPS by up to 27x with an average of 20.7x.

Index Terms—Graph Computation, High-Bandwidth Networks, High Performance, Computation Model, Communication Model

1 INTRODUCTION

Due to the wide variety of real-world problems that rely on processing large amounts of graph data [1], [2], many vertex-centric distributed graph-processing frameworks, including Pregel [3], GraphLab [4], PowerGraph [5] and GPS [6], 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 [3], [6], 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 batched communication aims to reduce the average communication overhead of the fine-grained messages [3], [6]. 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 [3], [6]. 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 [7], [8]. 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 [9], [10], [11] and severely underutilizes the network bandwidth capacity even when the *message buffering* technique [3], [6] 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 [12]. 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 e-

- *Y. Cheng and W. Guo are with College of Mathematics and Computer Science at FuZhou University, FuZhou 350016, Fujian, China. E-mail:{chengyongli, guowenzhong}@fzu.edu.cn*
- *H. Jiang is with Department of Computer Science & Engineering, University of Texas at Arlington, USA. E-mail:hong.jiang@uta.edu*
- *F. Wang, Y. Hua, D. Feng and Y. Wu are with the Wuhan National Lab for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, Hubei, China. E-mail:{wangfang, csyhua, dfeng, yxwu}@hust.edu.cn*

This is an extended version of our manuscript published in the Proceedings of IEEE International Conference on Computer Communications (INFOCOM), 2017, pages: 2340-2348.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

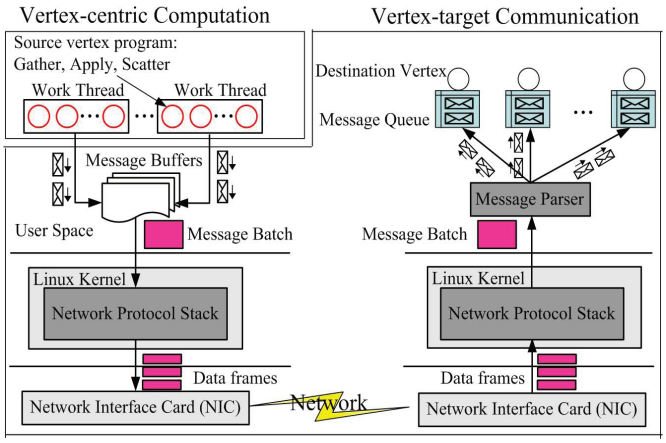


Fig. 1. A Pair of Compute Nodes in Vertex-Centric Distributed Graph-Processing Frameworks.

nough. 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 [13], 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 [3], [4], [5], [6], [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 [3], [9].

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 6, 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 [10].

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 [14], [15], [16], as discussed in Section 2.2. 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 4.

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 approach 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.* Based on the proposed computation and communication models, we implement a high-performance distributed graph-processing framework, called 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 2. The proposed computation model is given in Section 3. Section 4 introduces the proposed communication model. Section 5 presents other key components of the BlitzG. Experimental evaluation of the BlitzG prototype is presented in Section 6. We discuss the related work in Section 7 and conclude the paper in Section 8.

2 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 3. 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 4. We also introduce Data Plane Development Kit(DPDK), a high-performance user-space I/O framework. The key features of DPDK are mostly relevant to the key designs of our *light-weight message-centric communication model*, such as lockless design and reliable transmission.

2.1 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:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

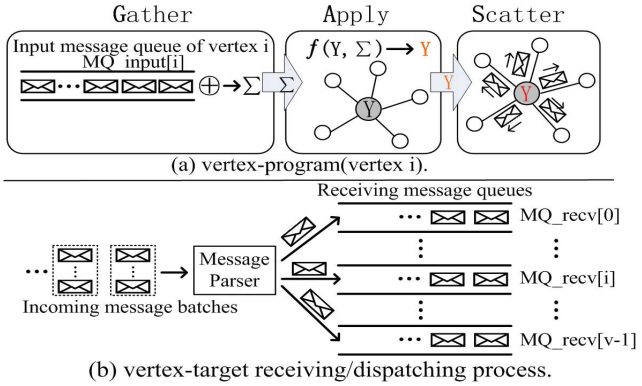


Fig. 2. The Workflow of the Vertex-Centric Distributed Graph-Processing Framework in A Typical Compute Node.

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

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 [4].

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 [3], [6] before sending them over the network.

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* [6]. 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***] [3], [6]. The former is used to store the messages that are sent to vertex ***i*** in current iteration. The latter stores the messages received in the previous iteration and serves as the input to the **vertex-program**(**vertex *i***) in current iteration. At the end of each iteration, the two message queues switch their roles.

2.2 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 [3], [6]. 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 [3], [6]. 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 [3], [6]. 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 [14].

Multi-copy overhead. In high-bandwidth networks, excessive data copying results in poor performance [17]. 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 [15], [16], [18], [19].

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 [16].

2.3 Data Plane Development Kit

The Linux networking functionalities are designed for general-purpose communication requirements. This general design is convenient for "normal" users. However, for many applications where fast packet processing is required, these functionalities rapidly reach a limit when running on a high-bandwidth network since the operating system cannot handle more packets and thus starts dropping them [20]. Data plane development kit (DPDK) is a fast user-space packet processing framework that can easily enable modern high-speed network devices (e.g., 10Gbit/s even 40Gbit/s Ethernet adapters) to work at line speed [15]. The high-performance of DPDK stems from the key features of *bypassing kernel*, *zero-copy*, *poll mode driver* and *huge-page memory allocations* [15], [17].

3 SLIMMED-DOWN VERTEX-CENTRIC COMPUTATION MODEL

3.1 Overview

We present the execution process of a typical compute node in our BlitzG framework to help understand our slimmed-down vertex-centric computation model.

Vertex workload-reduced computation process: 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*)** function. 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 Σ_input[***i***] is used to update the value of the vertex ***i*** (indicated as **Y**) :

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

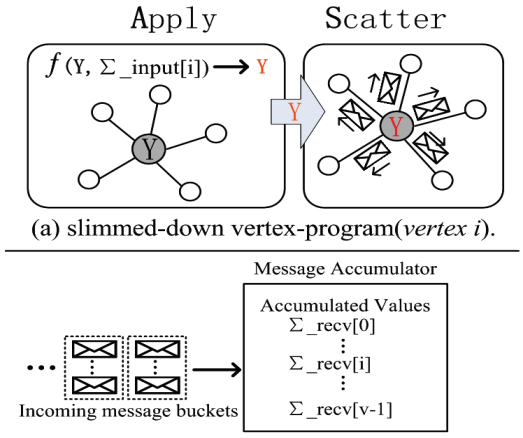


Fig. 3. The Workflow on A Typical Compute Node of BlitzG.

$$f(Y, \Sigma\_input[\boldsymbol{i}]) \rightarrow Y \tag{3.1}$$

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 4. When a message bucket is full, it is sent to the network interface card (NIC) directly.

Message-centric receiving/accumulating process: 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 4. 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:

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

where ***i*** is the destination vertex name of the message *msg*, and Σ_recv[***i***] is the receiving accumulated value of vertex ***i***.

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

3.2 Discussion

Like vertex-centric computation model in Pregel-like graph-processing frameworks, our light-weight message-centric communication model is also the part of our slimmed-down vertex-centric computation model. In order to fully exploit the high-bandwidth network capacity for fast graph computation, our slimmed-down vertex-centric computation model first reduces the computation workload of each vertex by moving the gather stage from the computation process to the communication process since, the messages are generated by the vertex programs in the computation process. Our slimmed-down vertex-centric computation model then employs the light-weight message-centric communication model to improve the communication efficiency significantly. This is important since, for the distributed graph-processing frameworks, the communication process is always the performance bottleneck due to the costly communication overheads, especially when a high-bandwidth network is available.

Although the amount of computation cannot be reduced.by moving the gather stage from the vertex program in compute process to the communication process. However, this design is an appropriate and necessary choice due to the reasons as follows. First, as mentioned before, since messages are generated by the vertex programs in the computation process, this design can improve the message generation speed by reducing the computation workload of each vertex. Second, more importantly, our communication model, with the added gather workloads, is efficient enough so that the communication process is not the performance bottleneck anymore in the high-bandwidth network, making it possible to fully exploit the high-bandwidth network capacity for fast graph computation by using the commodity compute nodes with limited core count.

Some existing distributed graph-processing frameworks, such as GPS [6] and MOCgraph [21], employ the receiver side combining technique that aims to reduce the memory footprint at the receiver side. However, our computation model is different from them in terms of the design goal and the effectiveness. First, the goal of our slimmed-down vertex-centric computation model is to fully exploit the high-bandwidth network capacity for fast graph computation and reduce the memory footprint. GPS and MOCgraph aim to reduce the memory footprint by using the receiver side combining technique. This can also reduce the workload of vertex program, improving the message generation speed. However, it is unmeaning since their performance bottle is the communication process, especially when a high-bandwidth network is available. Furthermore, by using the receiver side combining technique, the communication workload of these frameworks are added, aggravating the problem of communication bottle. This is a tradeoff between the performance and the reduced memory footprint.

4 LIGHT-WEIGHT MESSAGE-CENTRIC COMMUNICATION MODEL

4.1 Overview

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 Data Plane Development Kit (DPDK) [15], a fast user-space packet processing framework that has been gaining increasing attention [12], [15], [16], [19]. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

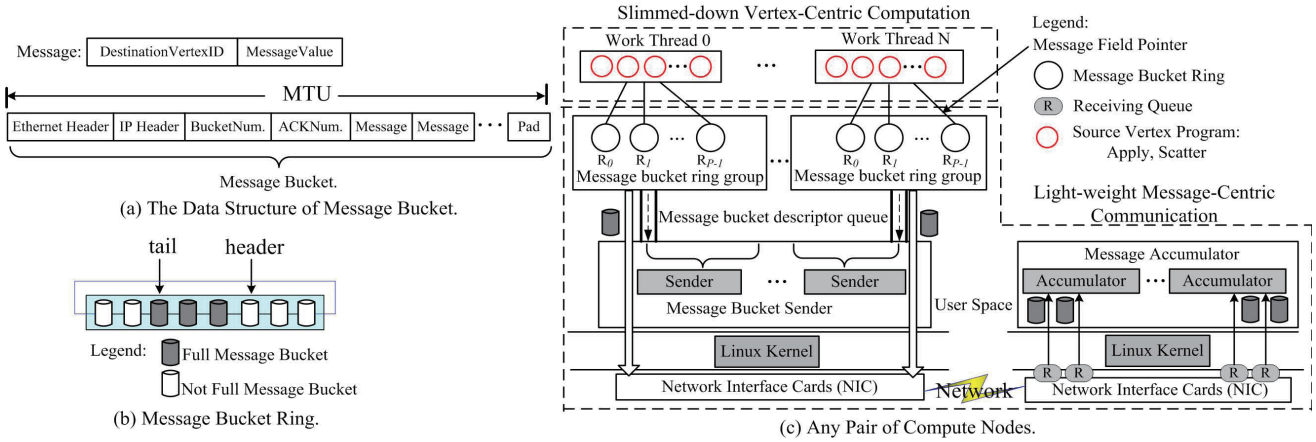


Fig. 4. The BlitzG Framework.

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 [15]. 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 [17]. 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 [15] 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 Σ_**recv[*i*]** variable, where ***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 reduce the average overhead of each message [3], [6]. 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 [22]. 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 [7], [8]. 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 5.1.

4.2 Discussion

In recent years, several general-purpose DPDK-based transport protocol, such as mTCP [23], have been proposed to provide reliable communication service for the DPDK-based applications. Like traditional TCP protocol, mTCP can provide services simultaneously for multiple DPDK-based applications due to the TCP header that includes the information of src/dst PORT numbers. Furthermore, the flows between two compute nodes can be routed on multiple paths if ECMP (Equal-Cost Multipath Routing) is used. This can help network switches to perform load balancing. The generality does, however, comes at a performance cost and the low utilization of memory space. For example, in order to provide services for multiple applications, large memory space is required by mTCP to buffer the intermediate data frames from the NIC, for different applications respectively. At the receiver side, the data frames are reorganized in the buffers according to correct data flow of each application, and then sent to the applications. The memory and the computation overheads of the protocol stack are too expensive to the communication-intensive and memory- hungry applications, such as BlitzG.

In order to address this problem, instead of using existing general-purpose DPDK-based transport protocol, we design the light-weight message-centric communication model specially for BlitzG, according the communication

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

features of distributed graph computation. This communication model removes the TCP/UDP headers away. At the receiver side, the *message accumulator* threads receive the message buckets from the NIC and process the messages in the message buckets directly, avoiding the overheads of memory and computation for the third-party protocol stack. This design has two limitations. The first one is that each NIC in a compute node can only service one application. Multiple NICs are required for each compute node to execute multiple applications simultaneously, leading to high hardware costs. However, this case usually rarely happen. For the communication-intensive and memory-hungry applications, system efficiency cannot be improved by executing this applications simultaneously. The second one is that all the flows between two compute nodes will be routed on a single path if ECMP used. However, this does not affect the load balancing in network switches. Since the communication model of BlitzG is all-to-all, that is, each compute needs to interact with each of the other compute nodes. Furthermore, the traffic load of each pair of compute nodes is nearly equal.

To sum up, for communication-intensive and time-sensitive DPDK-based applications, such as BlitzG and nginx [24], high performance and low overhead of memory can be obtained by designing specialized communication models for them. Due to the openness of DPDK, the design can be flexible according to the key features of the application for higher efficiency. Other DPDK-based applications can improve the performance by using general-purpose DPDK-based transport protocol. The generality does, however, comes at a limited efficiency.

4.3 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 directly on the fly, eliminating intermediate messages in vertex-centric distributed graph-processing frameworks. Memory consumption is an important concern in graph-processing systems [21]. 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.

5 DESIGN & IMPLEMENTATION

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 3 and 4. In this section, we focus mainly on other key components of BlitzG.

5.1 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 [16], [25]. 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 [12] with multiple queues that allow the packet receiving and processing to be load balanced across multiple processors or cores. For example, the Mellanox ConnectX-3 NIC has up to 32 queues [12]. 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.

5.2 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

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.

Delayed Sending Policy: In order to utilize the available network bandwidth capacity efficiently, each *sender* thread should send the *message bucket* to the NIC immediately when a *message bucket descriptor* is obtained from a *message bucket descriptor queue*. However, for any pair of compute nodes, the sender side should delay the sending process to slow down its sending rate when the network congestion occurs. Specifically, there is a time period between any two consecutive sending operations. The time period is amended dynamically according to the situation of congestion, which is defined as $AST \times N$. AST is the average time period of two consecutive sending operations, which is set to an initial value from the configuration file, and then amended during the execution process. N is a nonnegative integer that is set to 0 initially. During the execution process, N is increased by 1 according to the two cases that indicate the occurrence of congestion. First, the sender side has received three consecutive stale *AckNum* values. Second, the sender side does not receive any acknowledgement message within an expected time interval. Conversely, when three consecutive new *AckNum* values have been received, N is decreased by 1, with the condition of $N>0$.

A *NST*(next sending time) field in the *connection descriptor* helps the sender side to control the sending speed, which is set to the start time of the graph-computing job. Specifically, when a message bucket is ready to send, its *sender* thread needs to check the *NST* field. If the value of *NST* is later than the *current time*, the *sender* thread will abandon this sending operation, and continue to service other connections. Otherwise, the *sender* thread first sends the message bucket to the network interface card (NIC), and then updates the *NST* field by using the value of *current time* + $AST \times N$.

Discussion: The overheads of re-sending can be avoided by the coding techniques, such as erasure coding, which can recover the missing packets according to the coded information from other received packets. This is attributed to the DPDK that allows the user space applications to handle the NIC directly. The effectiveness of this method may be limited for the graph-processing systems that are designed for the high-bandwidth networks, such as BlitzG. Since the dropped packet rate is low in a high-quality network, there is a tradeoff between the avoided overheads of re-sending and the extra costs of the coding technique at both the sender and receiver sides.

However, this method may be useful for the graph-processing systems that run on the low-bandwidth networks. Since the overhead of re-sending is high in this case. However, this is non-trivial work due to the following reasons. First, the overhead of the coding technique should be low enough, compared with the avoided overhead of re-sending. Second, compared with the method of re-sending, the method of coding technique requires larger memory space in the receiver side to store the arrived and processed packets that are used to recover the missing packets. Third, if there are too many missing packets, the receiver side cannot recover them due to the incomplete coded information. In this case, it is important to select the right time for re-sending. Hence, it is meaningful to start a new study that can improve the efficiency of the reliable transmission in user-space graph-processing frameworks by using coding technique, in case of the high-bandwidth network is not available.

Summary: BlitzG can work independently without other transport protocols, such as TCP. Since, by using "sending with acknowledgement" mechanism and two policies of delayed processing and delayed sending, the reliable transmission between any pair of compute nodes can be guaranteed.

5.3 NUMA-based Performance Optimization

Existing distributed graph-processing frameworks routinely suffer from high communication costs. BlitzG overcomes the communication bottleneck by exploiting the high-bandwidth networks efficiently. Furthermore, today's network technology has developed so rapidly that high-bandwidth networks, such as 100Gbps Ethernet, are easily accessible than ever before. Hence, it is an opportunity for BlitzG to obtain higher runtime performance by improving the execution efficiencies of the involved threads. The higher efficiency of the involved threads can speedup the processes of message generation, sending, receiving and processing, leading to higher runtime performance. During the execution process of BlitzG, the involved threads need to access

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

memory frequently due to the fine-grained vertex-centric computation model. Hence, we further optimize the runtime performance by using a NUMA-based method as follows.

Fine-grained and NUMA-oriented data layout: The NUMA-oriented data layout is automatically configured when initializing the system. BlitzG first detects the number of nodes in the compute node, and then sets the data layout according to the number of nodes. BlitzG employs the fine-grained data layout to minimize remote accesses among the nodes. Specifically, each work thread executes its assigned vertices using 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. This is different from existing distributed graph-processing frameworks that use large buffer to hold the intermediate messages. Furthermore, each *bucket descriptor queue* is also managed independently by its *sender* thread. Hence, the graph data, including the vertices, *message bucket ring groups* and *bucket descriptor queues*, can be co-located to respective nodes according to work threads.

Minimized remote accesses: There are three key components in BlitzG, i.e., computation(work threads), *message bucket sender* and *message accumulator*. In each iteration, the work of each key component is parallelized by multiple dedicated threads, each of which works independently. In the computation component, each vertex *i* is executed by its work thread by using the *vertex-program*(**vertex *i***) that first reads the accumulated value Σ_input[***i***], and then generates one message for each of its neighbors by updating the *DestinationVertexID* and *MessageValue* fields of each message in the message buckets directly, requiring one read from and *di* writes to the memory, where *di* is the number of the outgoing neighbors of vertex *i*. In the *message bucket sender* component, as detailed in Section 5.1, the *sender* threads require one read from the memory before sending each message bucket to NIC.

Due to the fine-grained and NUMA-oriented data layout, each thread of the components of computation and *message bucket sender* can access the local memory of its node directly without any remote accesses, leading to higher execution efficiency. However, at the end of each iteration, the *message accumulator* component in each compute node needs to process the local message buckets that are sent to this compute node itself. When one message bucket is received, the value of each message *i* in the message bucket will be accumulated to the variable Σ_recv[***i***] that is located in the memory of one of other nodes possibly. In order to reduce the number of remote accesses, BlitzG first employs one local message queue for each node of the compute node. In each iteration, the work threads send each local message to its local message queue. This also incurs a remote access if this message belongs to one of the other nodes. In order to address this problem, BlitzG then introduces the 2-Level hierarchical partitioning to reduce the number of the remote memory accesses caused by the local messages.

Like most existing distributed graph-processing frameworks, BlitzG first distributes the vertices of the input graph to the compute nodes of the cluster by using a lightweight graph partitioning method, such as *Round-robin*. Unlike most existing distributed graph-processing frameworks, BlitzG further distributes the vertices in each compute node to the nodes of the NUMA architecture by using METIS [26], a high-quality graph partitioning method. Since the number of the vertices in each compute node is small, compared with the first level partitioning. Specifically, in each compute node, BlitzG first assigns the vertices to the nodes evenly according to the number of the nodes, and then employs the METIS to further improve the quality of the partitioning result. Due to the second level partitioning, the number of remote memory accesses of the *message accumulator* component can be reduced significantly, resulting in higher efficiency of each accumulator thread.

Discussion: The effectiveness of the NUMA-based optimization depends mainly on a key precondition that the performance of the graph-processing framework is dominated by the execution threads that access memory frequently. For example, Polymer [27] improves the performance of in-memory single-node graph-processing frameworks significantly by using the NUMA-aware optimization. Since the in-memory single-node graph-processing frameworks have no costly communication overheads. As a distributed graph-processing framework, BlitzG first overcomes the communication bottleneck, and then improves the performance by using the NUMA-based performance optimization, further obtaining $\sim$32% runtime improvement.

6 EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to evaluate the performance of BlitzG. Experiments are conducted on a network with leaf-spine topology that consists of leaf switches, spine switches and core switches. The Mellanox SN3000 switches are used, which is part of Mellanox's complete end-to-end solution, providing 10GbE through 200GbE interconnectivity within the data center. A 32-node cluster is used in these experiments. 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. The NIC of each compute node is connected to one of the leaf switches via a fiber cabling which can support up to 200Gbps bandwidth. The link speed between each compute node and its leaf switch is set to a value according to the NIC speed. Hence, in our experiments, the line speed refers to the NIC speed . We use 8GB of RAM for huge pages, based on the fact that in Intel's performance reports 8GB is set as a default huge page size [15]. The operating system of each node is CENTOS 7.0 (kernel3.10.0). DPDK-2.1_1.1 is used.

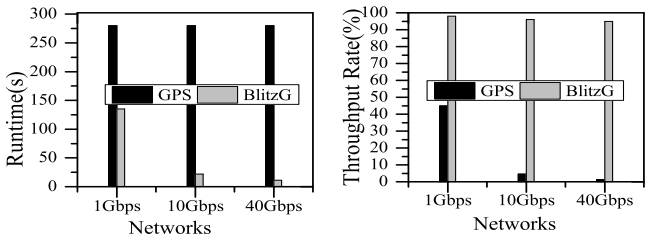
Graph Algorithms: We implement several graph algorithms to evaluate BlitzG by: Single-Source Shortest-Paths (SSSP) [30], PageRank (PR) [31], Community Detection (CD) [32] and Connected Components (CC) [32].

Baseline Frameworks: We compare BlitzG with two baseline frameworks. One is an up-to-date version of GPS, which is an open-source Pregel implementation from Stanfords InfoLab [6]. It is a representative BSP-based distributed graph-processing framework. The other is GraphLab, an open-source project originated at CMU [4] and now supported by GraphLab Inc. GraphLab is a representative distributed shared-memory graph-processing framework. We use the latest version of GraphLab 2.2, which supports

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

TABLE 1
Graph Datasets Summary.

DataSets	$\|V\|$	$\|E\|$	Type	Avg/In/Out degree	Max -/In/Out degree	Largest SCC
LiveJournal [28]	4.8×10^6	69×10^6	Social Network	18/14/14	20K/13.9K/20K	3.8M (79%)
Twitter-2010 [29]	41×10^6	1.4×10^9	Social Network	58/35/35	2.9M/770K/2.9M	33.4M (80.3%)
UK-2007-05 [29]	106×10^6	3.7×10^9	Web	63/35/35	975K/15K/975K	68.5M (64.7%)



(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).

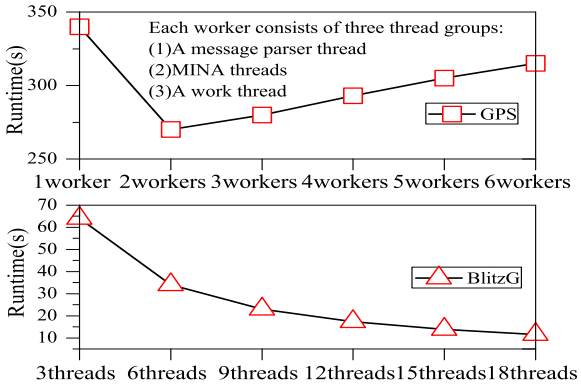


Fig. 6. Impact of Number of Threads.

distributed computation and incorporates the features and improvements of PowerGraph [5], [33].

Datasets: We evaluate BlitzG using three real-world graph datasets that are summarized in Table 1.

6.1 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 [6].

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 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).

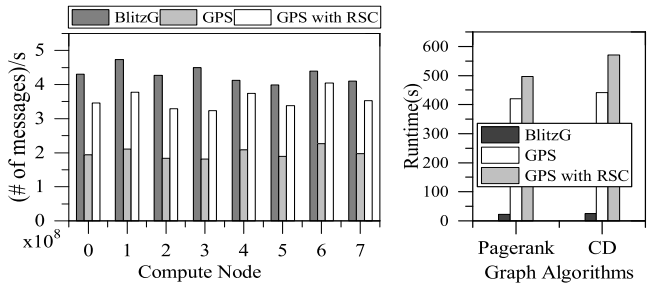
However, BlitzG obtains significant performance improvement on a higher-bandwidth network. The reasons are twofold. First, the slimmed-down vertex-centric computation model that significantly accelerates message generation by reducing the workload of each vertex. Second, the light-weight message-centric 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. We evaluate BlitzG comprehensively with different graph algorithms on various graph datasets against GPS. Experimental results indicate that BlitzG runs 17.2x-27.3x (with an average of 20.7x) faster than GPS respectively on various graph algorithms and graph datasets.

6.2 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 [6]. Each worker has a work thread, a message parsing thread and several MINA (an Apache network application framework) threads [6], [34]. 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 [35]. 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.

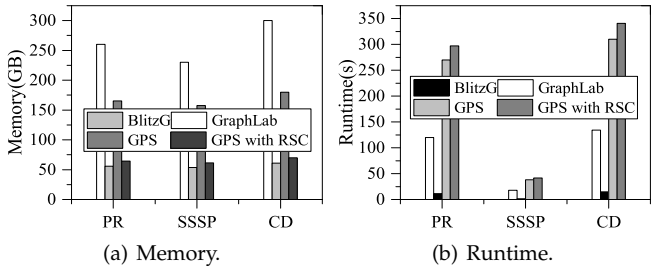
Unlike GPS, BlitzG uses multithreading in its three key modules of each compute node: computation (work threads), message bucket sender and message bucket accumulator. Instead of using a thread pool, BlitzG binds each thread to a dedicated logical core. BlitzG runs repeatedly by increasing the number 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems



(a) Message Generation Speed. (b) Runtime.

Fig. 7. Message Generation Speed& Performance Impact.



(a) Memory. (b) Runtime.

Fig. 8. Memory Consumption & Performance.

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 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.

6.3 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 five supersteps of 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 6.2, 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.

In order to obtain the message generation speed of each compute node, the number of total generated messages and the computation time should be obtained. Since the messages are generated by the work threads of the computation module, each loops through its vertices to generate messages. The generated messages are first sent to the message buffers in GPS (message bucket rings in BlitzG) where the message batches are then sent to the network. In this experiment, the computation time does not include any communication latency, due to the two reasons as follows. First, in both BlitzG and GPS, the computation and communication processes are executed in parallel. Once the computation module is activated, the sender module begins to receive and sent messages concurrently. Second, in order to avoid idle time experienced by work threads to wait for the communication, both the message buffers of GPS and the message bucket rings of BlitzG have a sufficiently large value.

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, 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 6.2, 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. In this experiment, the message generation speed of BlitzG is faster than that of GPS significantly. Experiments are also conducted with CC, CD and SSSP algorithms. Similar experimental results are obtained.

Experiments also are conducted to study the effectiveness of the receiver side combining technique in GPS. With the receiver side combining (RSC) configuration, GPS is run on an 8-node cluster executing five supersteps of the PageRank algorithm with the Twitter-2010. As shown in Figure 7, GPS with receiver side combining technique obtains 79% improvement over its original configuration in terms of message generation speed, but at the cost of a 9% performance loss in runtime. Experimental results indicate that GPS with receiver side combining technique can also improving the message generation speed significantly by reducing the workload of each vertex in the computation module. However, this gain is meaningless for GPS. In GPS, the performance bottleneck is the communication process, especially in a network ecosystem with high bandwidth. By using the receiver side combining technique, the communication workload is added, aggravating the problem of communication bottleneck.

6.4 Memory Consumption & Performance

MOCgraph [21] can achieve a similar performance to GraphLab [4] with significantly smaller memory consumptions. GraphLab is an open-source project originated at CMU [4] and now supported by GraphLab Inc. It is a representative distributed shared-memory graph-processing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems

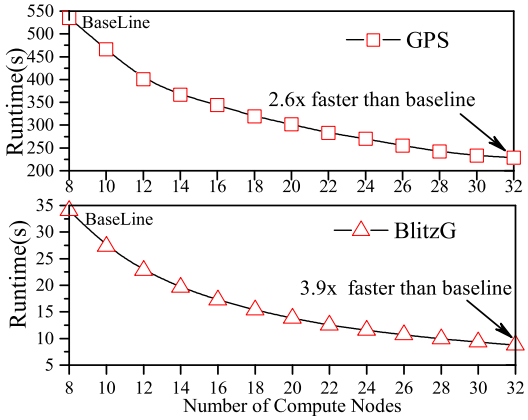


Fig. 9. Scalability.

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 [4], [5]. 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.

We also repeated experiments above to study the receiver side combining technique of GPS in terms of the memory consumption and performance. As shown in Figure 8, compare with its original configuration, GPS with receiver side combining technique obtains a memory footprint reduction of ∼61%. However, this is a tradeoff between the performance and the reduced memory footprint since the performance is also reduced in this case, as mentioned in Section 6.3.

6.5 Scalability

We conduct two sets of experiments to evaluate BlitzG against GPS in terms of the impact of the number of compute nodes. For each set of experiments, each framework runs 10 iterations of PageRank on the Twitter-2010 graph repeatedly, with the number of compute nodes ranging from 8 to 32. Note that each framework needs at least 8 compute nodes to run this graph-computing job. As shown in Figure 9, the runtime of BlitzG is reduced gradually by increasing the number of compute nodes. BlitzG runs 3.9x faster when running on a 32-node cluster than on an 8-node one. However, GPS runs only 2.6x faster when running on a 32-node cluster than on an 8-node one. These experimental results indicate that BlitzG has higher scalability than GPS. The high scalability stems mainly from the lockless design of the light-weight message-centric communication model. However, the high communication cost is most likely the key contributor to the weaker scalability of GPS.

TABLE 2
Reliable Transmission.

Thread Count (Computation)	Loss Rate (First Time)	Re-sending Times	Final Success Rate
1	0.06%	1	100%
2	0.09%	1	100%
3	0.13%	1	100%
4	0.11%	1	100%
5	0.12%	1	100%
6	0.19%	1	100%
7	0.13%	1	100%
8	0.09%	1	100%
9	0.13%	1	100%
10	0.14%	1	100%

6.6 Reliable Transmission

Experiments are conducted to study the effectiveness and efficiency of the "sending with acknowledgement" transmission mechanism. BlitzG runs 10 iterations of PageRank on the Twitter-2010 graph repeatedly by increasing the number of the work threads in each compute node, thus controlling the speed of message generation. In these experiments, we assign 4 threads to the message bucket sender, 6 threads to the message bucket accumulator. Experimental results, as shown in Table 2, indicate that there are very few dropped message buckets. The rate of the dropped message buckets is less 0.19%. By resending the dropped message buckets, 100% of the sent message buckets are received successfully. The network bandwidth capacity utilization (throughput rate) of the 40Gbps network increases with the number of work threads and reaches its peak value of 95%. These experimental results indicate that the "sending with acknowledgement" transmission mechanism is highly effective and efficient in providing transmission reliability.

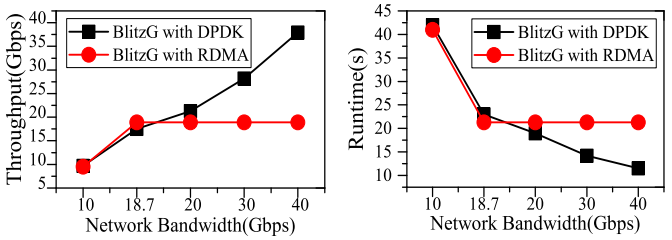
6.7 DPDK vs. RDMA

In recent years, several distributed graph-processing frameworks, such as Chaos [36], employs RDMA (Remote Direct Memory Access) technique [37] to provide high network bandwidth, aiming to reduce communication time. We compare RDMA with DPDK that is used in our light-weight message-centric communication model. For fair comparison, we implement the interaction between any pair of compute nodes by using RDMA technique. For RDMA, RoCE is used due to its higher performance, compared with iWARP. We call this version of BlitzG as BlitzG+RDMA. Each version of BlitzG, built on a 24-node cluster, runs the PageRank algorithm with the Twitter-2010 on the 10Gbps, 20Gbps, 30Gbps and 40Gbps Ethernets respectively.

Experimental results, as shown in Figure 10, indicate that BlitzG with DPDK obtains significant performance improvement on a higher-bandwidth network, achieving nearly line-speed throughput of each network. However, BlitzG with RDMA achieves the peak performance when the network bandwidth is limited to 20Gbps. A higher network bandwidth does not contribute to higher performances and throughput. When BlitzG with RDMA achieves its peak performance, the measured actual obtained bandwidth is 18.7Gbps.

These experimental results indicate that DPDK is notably superior to RDMA in terms of efficiency. This is also ev-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2875084, IEEE Transactions on Parallel and Distributed Systems



(a) Throughput. (b) Runtime.

Fig. 10. DPDK vs. RDMA.

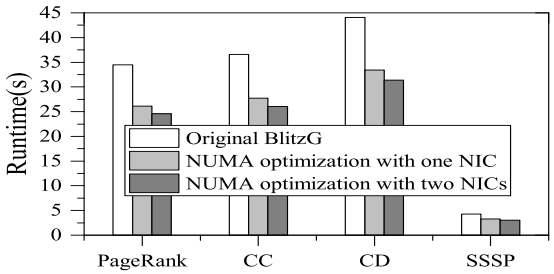


Fig. 11. Effectiveness of NUMA-based Optimization.

idenced by other related works [38]. There are likely two reasons as follows. First, compared with DPDK, RDMA moves the protocol stacks down to the NIC, depriving the limited computation resources of NIC. Second, by using DPDK, the workload of NIC is light. In order to improve the throughput of the network in the DPDK-based system, the effective solution is to add the core count dedicated to communication. In the DPDK-based system, the communication is light since the costly overheads of kernel, interrupt and data copying between kernel space and user space have been eliminated. Hence, DPDK is useful for the communication intensive applications, such as distributed graph-processing systems, due to its flexibility for users and high performance. RDMA is suitable for the applications that have a relatively light communication workload, but require lower communication latency, due to its user friendliness for users to implement their applications. Since both DPDK and RDMA have low communication latency.

6.8 Effectiveness of NUMA-based Optimization

Experiments are conducted on a cluster of 8 compute nodes to study the effectiveness of the NUMA-based optimization. Each compute node is with 64GB DRAM and two 6-core Intel Xeon E5-2620 processors. The whole memory are organized in two NUMA nodes, each has 32GB DRAM. In order to study the core affinity of the NICs, each processor node is directly connected to one 100Gbps Mellanox NIC using PCIe gen2. Access to hardware resources in the remote NUMA domain uses an interconnect between two processors.

BlitzG first runs PageRank with 10 supersteps with two configurations of original and NUMA-based optimization(one NIC) respectively. In these experiments, as shown in Figure 11 BlitzG with NUMA-based optimization(one NIC) obtains ∼32% runtime improvement over the original case. The performance gain stems from the NUMA-based optimization that reduces the number of the remote memory accesses significantly, resulting in higher execution efficiency of each thread of the three key components in BlitzG, i.e., computation, *message bucket sender* and *message accumulator*. In order to study the impact of the core affinity of the NICs, BlitzG then runs PageRank 10 supersteps with the NUMA-based optimization, by using the configuration of two NICs. In this experiment, BlitzG obtains ∼6% runtime improvement over the one NIC case. The reason is that, in the case of two NICs, each thread of the *message accumulator* component receives the message buckets from the NIC of its node respectively. However, in the one NIC case, the receiving threads in one node need to receive the message buckets from the NIC that has the better affinity to another node. Experiments are also conducted with CC, CD and SSSP algorithms. Similar experimental results are obtained.

7 RELATED WORK

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

Communication Efficiency: Pregel+ [39] develops two techniques to reduce the number of messages. The first technique is to create mirrors of each high-degree vertex, aiming to combine the messages of the high-degree vertex. However, since a mirrored vertex forwards its value directly to its mirrors, it loses the chance of message combining. Therefore, there is a tradeoff between vertex mirroring and message combining in reducing the number of messages [40]. The second technique is designed for pointer jumping algorithms where a vertex needs to communicate with a large number of other vertices that may not be its neighbors. This technique can prevent vertex r from receiving and sending a lot of messages, by combining all requests on each worker as one request towards vertex r [40]. GPS [6] introduces the dynamic repartitioning and large adjacency list partitioning (LALP) techniques to reduce the number of messages sent over the network. However, dynamic repartitioning also introduces extra network workload to reassign vertices among workers, leading to overhead that sometimes exceeds the benefits gained [30].

Memory Consumption: MOCgraph [21] 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. Giraph [9] serializes the edges and messages into byte arrays to reduce the number of objects, aiming to improve the memory utilization. Furthermore, a superstep splitting technique is developed to split a message-heavy superstep into several steps, so that the number of messages transmitted in each step does not exceed the memory size [40]. Like these techniques, BlitzG is memory-saving due to its light-weight message-centric communication model. However, Unlike these techniques, our communication model aims mainly to reduce the communication time of average message by avoiding the costly extra communication overheads, as mentioned before.

DPDK-based Applications: The DPDK framework is proposed recently to provide capacities of fast packet processing in software [15], which has been gaining increasing attention. In recent years, several large-scale internet services, such as DPDK-nginx [24] and DPDK-redis [41], have been transplanted to DPDK framework, aiming to provide

high-quality services by improving the communication efficiency. DPDK has also been used to the IoT (Internet of Things) system that typically encompasses a number of devices and sensors and is required to process a large number of messages at a high speed [42]. While BlitzG employs the DPDK technology to improve the communication efficiency in distributed graph-processing frameworks.

RDMA-based Graph-Processing Frameworks: Chaos [36] is a disk-based distributed graph-processing framework. Its system performance relies heavily on the assumption that network bandwidth far outstrips storage bandwidth [36]. In order to improve communication efficiency, Chaos employs RDMA (Remote Direct Memory Access) technique to provide high network bandwidth, aiming to reduce communication time. BlitzG is significantly different from Chaos because the former is designed to reduce the high communication costs experienced by existing in-memory distributed graph-processing frameworks while the latter aims to improve the performance of disk-based distributed graph-processing frameworks by reducing disk I/O and communication costs. GraM [13] improves communication efficiency by using the RDMA-based communication stack that preserves parallelism in a balanced way and allows overlapping of communication and computation. Unlike RDMA-based graph-processing frameworks, BlitzG employs the data plane development kit (DPDK) to speedup the communication process. DPDK is a fast user-space packet processing framework that can easily enable the high-speed network devices to work at line speed [15].

8 CONCLUSION

In this paper, we propose a highly efficient light-weight message-centric communication model that significantly reduces the "per-message" cost by fully exploiting modern high-speed networks. At the same time, we propose a slimmed-down vertex-centric computation model that not only significantly accelerates the message generation but also reduces the computation time. Based on the proposed new communication and computation models, we design and implement a high-performance distributed graph-processing framework, called BlitzG. Extensive prototype evaluation of BlitzG, driven by real-world datasets, indicates that it runs up to 27x (with an average of 20.7x) faster than GPS.

As future work, we plan to study the coding techniques that can be employed by graph-processing systems to reduce the overheads of the reliable transmission and fault tolerance.

ACKNOWLEDGMENTS

This work is supported by NSFC 61772216, 61672159 and U1705262. This work is also supported by Key Laboratory of Information Storage System, Ministry of Education and State Key Laboratory of Computer Architecture (No.CARCH201505), Technology Innovation Platform Project of Fujian Province under Grant (No. 2014H2005), Fujian Collaborative Innovation Center for Big Data Application in Governments, Fujian Engineering Research Center of Big Data Analysis and Processing.

REFERENCES

[1] T. Friedrich and A. Krohmer, "Cliques in hyperbolic random graphs," in *Proceedings of IEEE International Conference on Computer Communications*, 2015, pp. 1544–1552.
[2] X. Y. Li, C. Zhang, T. Jung, J. Qian, and L. Chen, "Graph-based privacy-preserving data publication," in *Proceedings of IEEE International Conference on Computer Communications*, 2016.
[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.
[4] 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*, vol. 5, no. 8, pp. 716–727, 2012.
[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *Proceedings of Usenix Conference on Operating Systems Design and Implementation*, 2012, pp. 17–30.
[6] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of ACM International Conference on Scientific and Statistical Database Management*, 2013.
[7] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.
[8] L. Page, "The pagerank citation ranking : Bringing order to the web," *Stanford Digital Libraries Working Paper*, vol. 9, no. 1, pp. 1–14, 1999.
[9] "Apache giraph." http://giraph.apache.org.
[10] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Proceedings of Usenix Conference on Operating Systems Design and Implementation*, 2012, pp. 31–46.
[11] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, and X. Wang, "Dd-graph: A highly cost-effective distributed disk-based graph-processing framework," in *Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 259–262.
[12] "Mellanox," http://www.mellanox.com/.
[13] 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, pp. 408–421.
[14] 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*, 2015, pp. 5–16.
[15] "Dpdk." http://www.dpdk.org.
[16] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *Proceedings of Usenix Conference on Networked Systems Design and Implementation*, 2014, pp. 445–458.
[17] L. Rizzo, "netmap: A novel framework for fast packet i/o." in *Proceedings of USENIX Annual Technical Conference*, 2012, pp. 101–112.
[18] 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.
[19] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of Usenix Conference on File and Storage Technologies*, 2012, pp. 3–3.
[20] D. Scholz, "A look at intel's dataplane development kit," *Network Architectures and Services*, pp. 115–122, 2014.
[21] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: scalable distributed graph processing using message online computing," *Proceedings of the Vldb Endowment*, vol. 8, no. 4, pp. 377–388, 2014.
[22] S. A. Athalye and T. Ji, "Method and apparatus for fragmenting a control message in wireless communication system," 2011.
[23] "mtcp," https://github.com/eunyoung14/mtcp.
[24] "Dpdk-nginx," https://github.com/ansyun/dpdk-nginx.
[25] 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 *Proceedings of IEEE International Conference on Parallel Processing*, 2015, pp. 330–339.
[26] G. Karypis and V. Kumar, "Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0," *Citeseer*, 1995.
[27] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 183–193.

[28] L. Backstrom, H. Dan, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 44–54.
[29] "The laboratory for web algorithmics." http://law.di.unimi.it/datasets.php.
[30] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: an experimental evaluation," *Proceedings of the Vldb Endowment*, vol. 8, no. 3, pp. 281–292, 2014.
[31] L. Page, "The pagerank citation ranking : Bringing order to the web," *Stanford Digital Libraries Working Paper*, vol. 9, no. 1, pp. 1–14, 1999.
[32] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Technical Report CMU-CALD-02-107, Carnegie Mellon University, Tech. Rep., 2002.
[33] "Graphlab." http://graphlab.org.
[34] "Mina." http://mina.apache.org/.
[35] M. Han, K. Daudjee, K. Ammar, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proceedings of the Vldb Endowment*, vol. 7, no. 12, pp. 1047–1058, 2014.
[36] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos:scale-out graph processing from secondary storage," in *Symposium on Operating Systems Principles*, 2015, pp. 410–424.
[37] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.
[38] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *Acm Sigcomm Computer Communication Review*, vol. 44, no. 4, pp. 295–306, 2014.
[39] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1307–1317.
[40] D. Yan, Y. Bu, Y. Tian, A. Deshpande *et al.*, "Big graph analytics platforms," *Foundations and Trends® in Databases*, vol. 7, no. 1-2, pp. 1–195, 2017.
[41] "Dpdk-redis," https://github.com/ansyun/dpdk-redis.
[42] J. G. Pak and K. H. Park, "A high-performance implementation of an iot system using dpdk," *Applied Sciences*, vol. 8, no. 4, pp. 550–566, 2018.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and PhD degree from Huazhong University of Science and Technology, Wuhan, China, 2017. He is a teacher of College of Mathematics and Computer Science at FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, FGCS, ToN and FCS.

Hong Jiang received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree from the University of Toronto, Canada, in 1987, and the PhD degree from the Texas A&M University, College Station, in 1991. He is Wendell H. Nedderman Endowed Professor & Chair of Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems and parallel/distributed computing. He serves as an Associate Editor of the IEEE TPDS. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACM-TOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, E-COOP, SC, ICS, HPDC, ICPP. Dr. Jiang is a Fellow of IEEE.

Fang Wang received her BE degree and Master degree in computer science in 1994, 1997, and Ph.D. degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 50 publications in major journals and conferences, including FGCS, ACM TACO, HiPC, ICDCS, HPDC, ICPP.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing and network storage. He has more than 80 papers to his credit in major journals and international conferences including IEEE TC, IEEE TPDS, USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS and IWQoS. He is a senior member of the IEEE, a member of ACM.

Dan Feng received the BE, ME, and PhD degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE-TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012.

Wenzhong Guo He received the BS and MS degrees in computer science, and the PhD degree in communication and information system from Fuzhou University, Fuzhou, China, in 2000, 2003, and 2010, respectively. He is currently a full professor with the College of Mathematics and Computer Science at Fuzhou University. His research interests include intelligent information processing, sensor networks, network computing, and network performance evaluation. He is a member of the IEEE.

Yunxiang Wu received the BE degree in computer science and technology from the Wuhan University of Science and Technology (WUST), China, in 2009. He is currently a PhD student majoring in computer architecture in Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and storage systems. He has several publications in major journals and international conferences, including IEEE-TPDS, ACM TACO and ICPP.