A Lightweight Communication Runtime for Distributed Graph Analytics

Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
{hdang8, brooks8, dryden2, snir}@illinois.edu

Roshan Dathathri*, Gurbinder Gill†, Andrew Lenharth‡¶, Loc Hoang§¶, Keshav Pingali**,¶
Department of Computer Science, ICES¶
The University of Texas at Austin
{*roshan, †gill, §loc, **pingali}@cs.utexas.edu, ‡andrewl@lenharth.org

***Abstract*—Distributed-memory multi-core clusters enable in-memory processing of very large graphs with billions of nodes and edges. Recent distributed graph analytics systems have been built on top of MPI. However, communication in graph applications is very irregular, and each host exchanges different amounts of non-contiguous data with other hosts. MPI does not support such a communication pattern well, and it has limited ability to integrate communication with serialization, deserialization, and graph computation tasks.**

In this paper, we describe a lightweight communication runtime called LCI that supports a large number of threads on each host and avoids the semantic mismatches between the requirements of graph computations and the communication library in MPI. The implementation of LCI is informed by lessons learnt from two baseline MPI-based implementations. We have successfully integrated LCI with two state-of-the-art graph analytics systems - Gemini and Abelian. LCI improves the latency up to 3.5× for microbenchmarks compared to MPI solutions and improves the end-to-end performance of distributed graph algorithms by up to 2×.

***Index Terms*—Graph analytics, clusters, communication systems**

I. Introduction

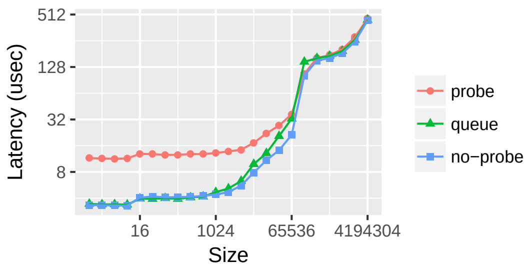
The performance of graph analytics applications on large-scale clusters is usually limited by communication. These applications are built using a variety of frameworks [1]–[10] that are in turn implemented on top of TCP or MPI. MPI can provide better performance than TCP, but it is not an ideal communication interface for graph analytics. Indeed, applications in this domain are quite different in their behavior from the scientific computing applications that MPI was designed for, for the following reasons:

- Graph analytics applications are less compute-intensive than typical scientific applications; in fact, clusters are used in graph analytics for their large memory rather than their computational capability. This makes it difficult to overlap communication with computation to reduce the performance impact of communication.
- Graphs that arise in traditional HPC applications are often uniform-degree graphs, and their total size increases polynomially in their average diameter. In contrast, graphs of interest in graph analytics are power-law graphs, and their total size is exponential in the average diameter [11]. For many problems, it is necessary to use different parallel algorithms for these two classes of graphs [12].
- Most traditional HPC applications are *topology-driven* in which identical operations are performed on each node of a graph. In contrast, efficient graph analytics algorithms are *data-driven algorithms* in which computations are performed only at certain *active* nodes in the graph. Nodes become active in data-dependent, statically unpredictable ways, so the patterns of control-flow and data accesses are much more irregular [13].

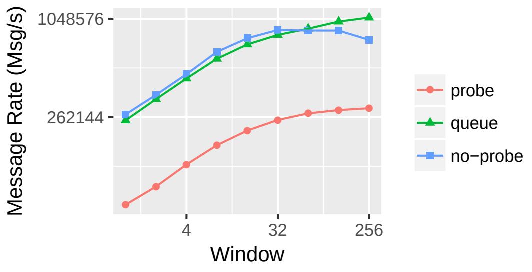
Because of these factors, an efficient communication system is even more critical for good end-to-end performance of graph analytics applications than it is for traditional HPC applications. In the HPC context, new Network Interface Cards (NIC) are being developed by companies such as Mellanox and Intel. These adapters support remote direct memory access (RDMA) natively and provide high communication rates of millions of messages per second. Vendors are starting to develop new standard interfaces for working with these new devices, such as UCX from IBM/Mellanox [14] and OFI from Intel and other companies [**?**]. Vendors are also adding features in their adapters to accelerate MPI communication.

Nevertheless, MPI semantics are not the best match to the needs of graph analytics computations. Some MPI features, such as its strict message ordering requirements or its support for don't-care receives, are known to be impediments to high message rates, especially with many concurrent communications [16]. The problem is partly due to the difficulty of implementing fine-grained synchronization in a complex concurrent code without slowing down the uncontested path and partly intrinsic to the design of MPI which forces the traversal of sequential lists [17]. While improved support for multithreading with MPI is certainly possible [18], [19], performance still tapers off with large thread counts. This problem worsens when each host communicates simultaneously with many other hosts (resulting in many concurrent pending receives) and when each host is running many threads. The Intel Xeon Phi Knights Landing (KNL) has 68 cores and is capable of running up to 272 hardware threads simultaneously. Future systems will have larger numbers of threads.

In addition, MPI does not efficiently support receiving messages of unknown size: one either needs to allocate buffers



(a) Latency benchmark with increasing data size



(b) Message Rate benchmark with increasing message window size for 64-byte message

Fig. 1: Performance comparison of MPI_SEND/RECV (no-probe), MPI_PROBE (probe) and LCI (queue) using an adapted OSU latency and message rate test [15]. See Section IV for the experimental setup.

of maximum message size or add a probing call (`MPI_PROBE` or `MPI_IPROBE`) to find out message size. MPI also does not provide support for message-driven scheduling of threads; this is usually implemented by a polling agent that sits on top of MPI, separate from the polling done by the progress engine of MPI. Finally, MPI was designed to be used directly by application developers, not by framework or language implementers. A framework implementer may prefer a lower-level interface with more control and predictable performance.

This paper studies the performance advantages that accrue in graph analytics applications by shifting from MPI to a new communication runtime called *Lightweight Communication Interface* (LCI). LCI maps more directly to the underlying hardware, provides more lightweight interaction with threads, and avoids overheads from semantic features of MPI not required for graph analytics applications. Our evaluation on an Intel KNL cluster connected by Intel Omni-Path NICs shows that applications run significantly faster on LCI than on MPI, and they scale well to large thread counts per host on LCI. We also show that LCI and its performance is also portable to Infiniband NICs and consistently better in comparison to three state-of-the-art MPI implementations. Figure 1 shows the improvement with a typical latency and message rate benchmark using three different interfaces: using `MPI_SEND/MPI_RECV` (no.probe), using `MPI_PROBE` at the receiving side (probe), and using LCI for sending and receiving messages (queue). It shows that using LCI significantly reduces the overhead of the communication by up to a factor of 3.5$\times$ in comparison to probe.

The rest of this paper is organized as follows. In Section II, we describe the two state-of-the-art graph analytics systems used in our study, Gemini [7] and Abelian (as yet unpublished), which is a distributed-memory version of the Galois system [20]. Both have similar communication patterns. Section III describes the communication libraries based on MPI and LCI and explains the key design choices made in each implementation. Section IV describes experimental results using both Abelian and Gemini. Related work is described in Section V. We summarize the main conclusions in Section VI.

II. Distributed-memory Graph analytics

The Gemini and Abelian systems support *vertex programs*: some of the nodes in the graph are initially *active*, and applying an *operator* to an active node makes it inactive and may make some of its neighbors active. An operator can only access the labels of the active node and its immediate neighbors. A *push-style* operator reads the active node's label and writes its neighbors' labels, and a *pull-style* operator reads its neighbors' labels and writes the active node's label. Computation terminates when all nodes are quiescent.

On distributed-memory clusters, the graph is partitioned among hosts using one of many partitioning policies. Gemini only supports the edge-cut partitioning strategy whereas Abelian supports general partitioning strategies including edge-cuts and general vertex-cuts. A simple way to think about these partitioning strategies is to consider a partitioning of edges to hosts. If an edge (u, v) is assigned to a host, the host creates *proxies* for nodes u and v and connects them with an edge. Since the edges connected to a given node in the graph may be partitioned among several hosts, it is possible for a node to have many proxies in the partitioned graph. One of these proxies is designated the *master* proxy while the rest are designated as mirror proxies. The master proxy is responsible for the canonical value of the vertex for which it is a proxy.

Since the proxies for a given graph node may be read and written by different hosts, we need a synchronization strategy to coordinate the reads and writes. One approach is to use distributed shared-memory (DSM) [8] but the overheads of this approach are substantial. Instead, Abelian and Gemini use the Bulk-Synchronous Parallel (BSP) model for synchronization [21]. The program is executed in rounds, and each round consists of computation followed by communication. During the computation phase, each host applies the operators to active nodes in its partition. The communication phase is used to synchronize the labels of all proxies, and it can be composed from two patterns. The first, *reduce*, has all mirror proxies communicate their values to the master proxy, where the master proxy combines them into a canonical value. The second, *broadcast*, has the master proxy communicate the canonical value to all mirror proxies. Depending on the partitioning policy used as well as the operator (push or pull), reduce, broadcast, or both are necessary to synchronize the required values.

The Abelian runtime is partition-aware. It minimizes the communication volume by choosing reduce, broadcast, or both, based on the partitioning policy. It also minimizes the communication meta-data while synchronizing only the updated labels, thereby further reducing communication volume. For these reasons, we choose to demonstrate our communication runtime by using Abelian for the most part. Although Gemini supports only edge-cut partitioning, it is the state-of-the-art distributed graph analytics system, so we use it too in our studies.

III. Communication: MPI and LCI

This section describes the way we implement communication in Abelian programs. Section III-A shows that the inter-host communication required in each round of execution of an Abelian program can be described abstractly as a *gather-communicate-scatter* communication pattern. Sections III-B and III-C describe how this communication pattern is implemented using MPI-Probe and MPI-RMA respectively. Section III-D describes LCI - our customized communication layer for this pattern. The communication pattern in Gemini is a special case of this since it supports only edge-cuts.

A. Communication in Abelian programs

As discussed in Section II, there are two communication patterns of interest: (1) reduce from mirrors to their master, and (2) broadcast from a master to its mirrors.

To understand how the communication pattern is implemented, it is necessary to consider the in-memory representation of graphs in the Abelian system. On each host, the master nodes are stored contiguously, followed by mirror nodes. The data for each graph node is usually a struct with several labels or fields and this data is stored as an array of structs (AoS).

Not all nodes are active at the same time, and not all labels need to be communicated. Thus, the data to be communicated from one host to another is not contiguous, neither on the send side, nor on the receive side. The layouts of communicated data in sender memory and receiver memory are not identical. Furthermore, what data is communicated changes at each round, and is data-dependent. Sending each entry in an individual message is not practical, nor is it feasible to use `MPI_REDUCE`. In general, each host may need to communicate with each other host. Each send to another host is preceded by a *gather* operation that stores in a contiguous buffer the data to be sent to that host; each receive from another host is followed by a *scatter* operation that retrieves data from the contiguous receive buffer and updates the relevant entries. The update involves copying the data if the communication is a broadcast, or applying a reduce operation, if the operation is a reduce. The gather and scatter patterns may involve different sets of hosts at each round. Thus, both cases result in a *gather-communicate-scatter* pattern of communication. Note that the same communication pattern holds for other distributed memory graph frameworks.

If a host needs to communicate values to m other hosts, it is necessary to perform m gather operations. These gather operations are independent and in principle can be performed in parallel. Scatter operations can also be parallelized, taking care to avoid races for the reduce updates.

This gather-communicate-scatter pattern is implemented in Abelian as follows. One thread on each host is dedicated to communication with other hosts. The other threads perform computation during the local computation phase, but they also participate in communication during the communication phase; to keep the terminology simple, we refer to them nevertheless as *compute threads*.

Figure 2 shows an overview of the Abelian communication runtime on each host during one round of communication. The compute threads perform gathers into buffers in parallel. Completed buffers are enqueued to the send queue of the communication thread. Once gathers are complete, the compute threads switch to performing scatters in parallel to process messages received from other hosts. The messages from other hosts are processed in an arbitrary order as they arrive. The dedicated communication thread interleaves sending and receiving. It checks to see if there are buffers that need to be sent, and if so, it pushes them out into the network. It also polls the network for incoming messages, and enqueues them into a receive queue to be processed by the compute threads. To maximize throughput, no blocking operations are used.

This design is intended to reduce the overhead of communication by parallelizing gathers and scatters, and overlapping them with communication. In particular, the communication thread can be sending and receiving data while the compute threads are performing gathers and scatters. This design is also intended to minimize the latency. The compute threads enqueue a message to a host without waiting to prepare messages for all hosts and they dequeue messages from any host as and when they arrive without waiting to receive it in some order. Similarly, the dedicated thread sends and receives messages as soon as it can, without prioritizing any host. The performance of this communication strategy depends on how well the network layer can handle the irregular communication and minimize the latency.

In the next three subsections, we present three approaches to tackle this issue: using traditional MPI two-sided communication (MPI-Probe), using MPI3 one-sided features (MPI-RMA) and finally our low-level communication layer (LCI) which attempts to mitigate the drawbacks of the previous two approaches.

B. MPI-Probe communication layer

This section describes the two-sided MPI-based implementation of the Abelian communication layer. It uses an additional buffered network layer. This is the baseline implementation against which we compare our optimized runtime.

As mentioned in Section III-A, there is a dedicated thread for sending and receiving messages. The `MPI_THREAD_FUNNELED` mode is used, and MPI commands are only issued from the dedicated communication thread. This is important for performance; although MPI provides concurrent accesses (`MPI_THREAD_MULTIPLE`), currently deployed implementations are known to suffer substantial performance loss when `MPI_THREAD_MULTIPLE` is used [16], [18], [22].

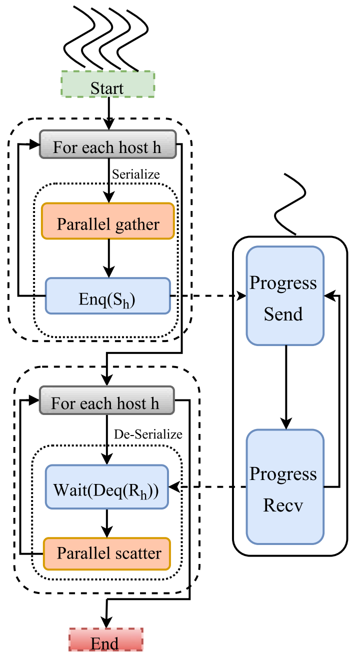


Fig. 2: Communication in the bulk-synchronous phase in Abelian: gather/serialization and scatter/deserialization are done by many threads (column on the left); the communication buffer is submitted to the communication thread for actual communication; the two methods `Enq` (enqueue) and `Deq` (dequeue) are non-blocking and used for issuing a buffer or testing for incoming buffer to/from the communication thread.

A naive implementation of Abelian on top of MPI reveals one basic problem of the MPI communication model when non-blocking communication is used: the lack of back pressure on producers when data is produced faster than consumed. This is especially problematic with MPI's eager protocol, as it may lead to the exhaustion of MPI data buffers. The slow consumption can occur either at the network interface due to packet injection rate limits on many networks, resulting in buffer exhaustion on the sending side, or at the consumer, resulting in buffer exhaustion on the receiving side. The lack of back pressure also increases buffer consumption at the application level, because of the all-to-all communication pattern. For these reasons, the communication pattern of Abelian may cause MPI to either seg-fault or hang due to unrecoverable errors from the network devices or from the software implementation (we tested with MVAPICH2 and Intel MPI).

To alleviate this problem, we added a buffered network layer that works as follows. For sending messages (maybe different datatypes or fields), the system buffers small items (those less than the eager-send limit) until either the oldest buffered message times out or the buffer size exceeds the eager send limit. This reduces buffer consumption (data is not copied and buffered) while capping latency. Buffering is done via thread-safe multi-producer, single consumer queues, which implement the `Enq` and `Deq` operations from Figure 2. The communication thread pops from this queue and only interacts with MPI serially, which minimizes the thread synchronization overheads and controls the memory usage of MPI. The communication thread uses `MPI_ISEND` for sending messages without blocking.

For receives, `MPI_IRECV` cannot be used directly because communication is irregular and dynamic (*e.g.*, there is no prior information about the incoming message size). Abelian makes use of `MPI_IPROBE`[1] with MPI wildcards to handle incoming data. The `MPI_STATUS` returned by this function provides the information to start receiving the message, such as the size of the buffer, the source and tag of the message. Subsequently, a `MPI_IRECV` is called to continue with the pending communication. The communication thread uses `MPI_TEST` calls to ensure forward progress and reclaim buffer space as communications, both sends and receives, are complete. All MPI calls are non-blocking to allow multiplexing between resources and avoid resource exhaustion.

C. *MPI-RMA communication layer*

This section describes the implementation of the Abelian communication layer based on one-sided MPI. It adapts the communication runtime shown in Figure 2 to perform MPI Remote Memory Access (RMA) operations. The goal of the MPI RMA implementation is to establish a lower-bound for communication overheads since, by using MPI RMA, we can avoid two-sided matching of send/recv and also avoid thread synchronization between computation and communication thread. However, as we will see, this comes with a cost of increasing the overall memory usage.

Typical MPI RMA implementations [23] are single-threaded, store the entire graph in a RMA window, and access the graph during computation via MPI RMA calls, thus limiting computation efficiency and the opportunity for communication aggregation. In contrast, we create RMA windows only for receiving aggregated messages during communication so that the computation accesses only the local graph. The main challenge in achieving this is in determining the receive buffer sizes since they need to be pre-allocated. As mentioned earlier, in graph analytics application, the data communicated in each round of communication varies widely, even between the same pair of hosts. However, an upper bound can be computed assuming all nodes are active. This can be higher than the average.

For a particular host, all hosts determine the maximum size of the message that they can receive from that host and allocate a buffer of that size in the window collectively (`MPI_WIN_CREATE`). In other words, for p hosts, there are p shared windows, each of which have $p-1$ remote buffers. Such a set of windows is created for each datatype that is communicated (on first communication) for each pattern of communication (reduce and broadcast).

[1]We did not use `MPI_IMPROBE` / `MPI_MRECV` since all communication is done by one thread and, in our experiments with both MVAPICH2 and IntelMPI, `MPI_IMPROBE` / `MPI_MRECV` is slower and/or hang with various benchmarks.

Since the communication phase is bulk-synchronous in Abelian, passive target synchronization on RMA windows is not suitable, so we implement active target synchronization. One way to achieve that is to use a collective synchronization model on the windows (`MPI_FENCE`). However, such synchronization is too restrictive since it has to wait for all RMA operations on all hosts to complete, thereby hurting the performance. Since our goal is to reduce the latency, we choose to implement a generalized active target synchronization, which allows fine-grained synchronization.

In the MPI-RMA communication layer, the main compute thread, shown in Figure 2, which would enqueue and dequeue send and receive buffers will instead perform RMA operations. To send messages, a host will start an access epoch on its RMA window (`MPI_WIN_START`). For each destination, a parallel gather by all compute threads is performed. This prepares the send buffer (source data), which is then written using `MPI_PUT` to the remote memory or buffer of that destination in the host's window. Finally, after the `MPI_PUT` is initiated for all remote destinations, the host will complete the access epoch on its window (`MPI_WIN_COMPLETE`). A host exposes its receive buffers (in different windows) at the end of a round with calls to `MPI_WIN_POST` and checks for the completion of the remote access to its receive buffers with calls to `MPI_WIN_WAIT` after completing the access epoch on its window. When the `MPI_WIN_WAIT` for a remote source's RMA window returns, the host performs a parallel scatter to process the (local) received buffer and then posts a new exposure epoch on the source's window using `MPI_POST`.

Unlike in Figure 2, the dedicated communication thread in the one-sided MPI communication layer does not interact with the computation threads. However, the dedicated communication thread continuously polls the network (`MPI_IPROBE`) to ensure forward progress [24] for the MPI RMA operations[2]. Since both the main compute thread and the dedicated communication thread are issuing MPI commands, this layer uses `MPI_THREAD_MULTIPLE`.

D. LCI communication layer

In this section, we describe the LCI implementation of the Abelian communication layer. LCI not only reduces the latency of messages when compared to two-sided or one-sided MPI implementations, but also dynamically manages memory requirements, thereby reducing memory usage compared to the one-sided MPI implementation.

We present the design of **Queue**, an LCI interface for supporting Abelian and similar irregular communication patterns. The goal of Queue is to avoid the short-comings of MPI implementations described above.

- *LCI avoids fatal failures due to insufficient network resources*. This is done by allowing the upper layer to retry the operation on such events. The MPI standard does not require implementations to handle resource exhaustion errors and in current MPI implementations the program crashes when these happen. This problem has to be mitigated by an additional buffered layer as discussed before.
- *LCI supports multi-threading efficiently with a communication server*. The interaction between the server and the compute thread is limited to a single flag. This is not possible in MPI; a `MPI_TEST` leads to an expensive network poll, thus leading to additional operations for checking request completion.
- *LCI is closer to the network interface*. This prevents any buffering and duplicated functionality due to the complexity of the MPI implementation (e.g. there is no tag-matching or ordering enforcement in the LCI interface).

[2]Another possible option is to use the MPI asynchronous progress thread, but it is more heavyweight and we have less control over the thread.

Algorithm 1 SEND-ENQ operation (executed by thread)

```
1: $P$: a global concurrent packet pool.
2:
3: procedure SEND-ENQ($b, s, h, t$)   ▷ : buffer, size, rank, tag
4:     $p$ = packetAlloc($P, s, h, t$)
5:     if $p$ then
6:         $r$ = makeRequest($p$)
7:         if $s$ is small then
8:             copy($p.b \leftarrow b$)
9:             $p.type \leftarrow$ EGR
10:            lc_send($p$)
11:            $r.status \leftarrow$ DONE
12:        else
13:            $r.status \leftarrow$ PENDING
14:            $p.src \leftarrow b$
15:            $p.type \leftarrow$ RTS
16:            lc_send($p$)
17:        end if
18:        return $r$
19:    end if
20:    return NULL
21: end procedure
```

Communication in LCI involves the following two steps.

Communication Initiation: Communication is started by obtaining resources for sending data or checking if there is an incoming packet to process. When successful, it returns a request handle which contains a record of the communication and the resources corresponding to the communication. In comparison to an MPI non-blocking function (such as `MPI_ISEND`), our initiation can fail if there are no available resources (for sender) or there is no pending communication (for receiver). However the failure is not fatal and simply means the user should retry at a later time. The two functions for initiation of send and receive are **`SEND-ENQ`** and **`RECV-DEQ`** respectively.

Communication Completion: After initiation is successful, the communication is now in progress. The progress is implicit and typically ensured by a communication server. When the communication is finished, a boolean flag is set. In comparison

Algorithm 2 RECV-DEQ interface (executed by thread)

```
 1: $Q$: a global concurrent queue.
 2: $P$: a global concurrent packet pool.
 3:
 4: procedure RECV-DEQ($*b, *s, *h, *t$)
 5:                         ▷ : pointer to buffer, size, rank, tag
 6:     $p$ = dequeue($Q$)
 7:     if !$p$ then                                ▷ $Q$ is empty
 8:         return NULL
 9:     end if
10:     $r \leftarrow$ makeRequest($p$)
11:     $(*s, *h, *t) \leftarrow p.header$
12:     $*b \leftarrow$ allocate($*s$)
13:     if $p$ is EGR then
14:         copy($*b \leftarrow p.b$)
15:         $r.status \leftarrow$ DONE
16:         packetFree($p$)
17:     else
18:         $p.dst \leftarrow *b$
19:         $r.status \leftarrow$ PENDING
20:         $p.type \leftarrow$ RTR
21:         lc_send($p$)
22:     end if
23:     return $r$;
24: end procedure
```

Algorithm 3 Network progress (executed by server)

```
 1: $Q$: a global concurrent queue.
 2: $P$: a global concurrent packet pool.
 3:
 4: procedure NETWORK-PROGRESS
 5:     $p \leftarrow$ lc_progress
 6:     if $p.type$ is EGR or RTS then
 7:         enqueue($Q, p$)
 8:     else if $p.type$ is RTR then
 9:         $p.type \leftarrow$ RDMA
10:         lc_put($p$, $p.src \rightarrow p.dst$)
11:     else if $p.type$ is RDMA then
12:         $p.r.status \leftarrow$ DONE
13:         packetFree($P, p$)
14:     end if
15: end procedure
```

to MPI functions such as `MPI_TEST` or `MPI_WAIT`, our mechanism is more lightweight: there is no need for a function call; the user maintains a list of requests and checks the status flag fields.

To implement Queue, we make use of some abstractions for interacting with the underlying network APIs. We present here a simplified version of this list of functions:

- **lc_send**(p): submit a command to the network which transfers a limited-size packet structure (p) enclosing a header with some information such as a rank and tag of the destination, the type of the packet, and some data. Every host has to maintain a fixed number of buffers for receiving these packets.
- **lc_put**(p, $src \rightarrow dst$): submit a command to the network which transfers data from a source buffer (src) to a target buffer (dst), identified by a host and key for address translation enclosed in the packet p.
- **lc_progress**(): ensure progress of the communication such as flushing outgoing data and peeking for an incoming packet. If a packet is received, from any host, the function returns it to the caller.

`lc_send` and `lc_put` are non-blocking and are typically very short. They can be executed by both communication and computation threads. `lc_progress` can take longer since it typically requires draining the network driver by executing the network progressing functions. Hence, it is only executed by the communication thread. `lc_send` is provided by most network interface APIs and is typically used for short messages in an eager protocol. `lc_put` can be implemented directly by the hardware if the network interface supports RDMA. In particular, for *psm2*, the native network API of Omni-Path, `lc_put` is implemented by translating target identification to a special tag. This is convenient enough since *psm2* has a rich set of tag-matching interfaces (96 bits can be used for matching purposes). On the other hand, for *ibverbs* of Infiniband devices, our implementation using reliable connection (RC) is straightforward: both `lc_send` and `lc_put` map directly to `ibv_post_send` calls using `IBV_WR_SEND` and `IBV_WR_RDMA_WRITE` work request respectively [3].

The pseudocode for send and receive with the Queue interface is presented in Algorithms 1 and 2 for both eager and rendezvous protocols (selected automatically depending on the size of the incoming buffer). A request is a structure for storing the ongoing communication status and ties to a packet for flow control. The rest of the communication is done by the communication server as presented in Algorithm 3. The basic idea is for the communication server to progress the network and execute appropriate callbacks for each packet type. Specifically, packet types are as follows: **EGR** - eager packet for short protocol which includes the data; **RTS, RTR** - ready-to-send and ready-to-receive control packets respectively, which are commonly used for the rendezvous protocol to exchange the buffer addresses; and **RDMA** - packet specifically for the `lc_put` operation.

The algorithms also rely on two variables P and Q which are accessed atomically for supporting thread-safety: *packetAlloc/Free* - to allocate/free a packet; *enqueue/dequeue* - to store/retrieve incoming packets. These operations can be easily implemented with a concurrent pool and a concurrent queue respectively. We implemented the locality-aware packet pool presented in [16] and the fetch-and-add based MPMC queue presented in [26]. The size of the packet pool determines the

[3]RC is sufficient for our current purpose since we maintain one process per host. One can also emulate RDMA atop other connection types like in [14], [25].

TABLE I: Inputs and their key properties.

	clueweb12	kron30	rmat28
$\|V\|$	978M	1073M	268M
$\|E\|$	42,574M	10,791M	4,295M
$\|E\|/\|V\|$	44	16	16
max D_{out}	7,447	3.2M	4M
max D_{in}	75M	3.2M	0.3M

TABLE II: Total execution time (seconds) for Abelian at 128 hosts using the rmat28 graph.

	Stampede2			Stampede1		
	LCI	MPI-Probe	MPI-RMA	LCI	MPI-Probe	MPI-RMA
bfs	0.59	0.60	0.62	0.50	0.52	0.55
cc	0.95	1.44	1.21	1.12	1.15	1.21
pagerank	17.60	44.26	33.21	22.05	23.09	27.65
sssp	1.11	1.17	1.11	1.09	1.12	1.24

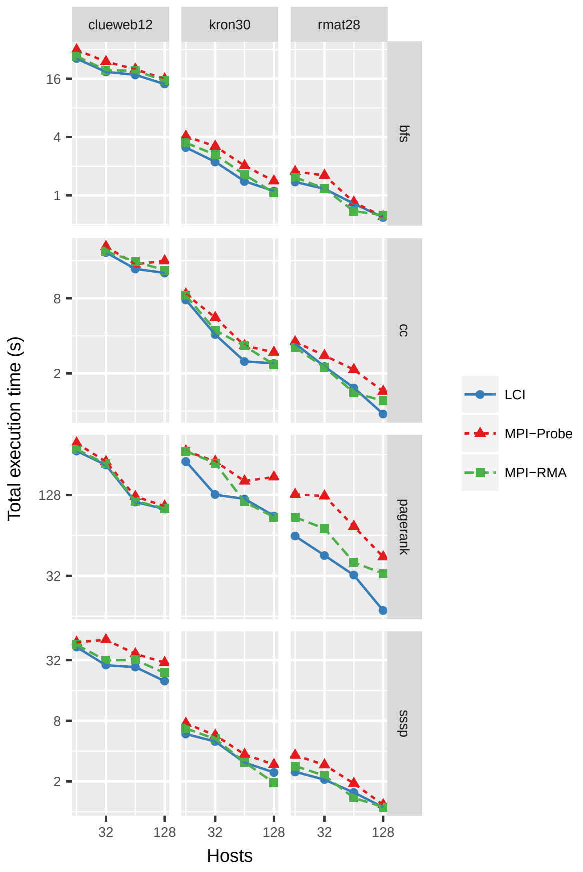


Fig. 3: Total execution time on Stampede2: Abelian with LCI, MPI-Probe, and MPI-RMA runtimes.

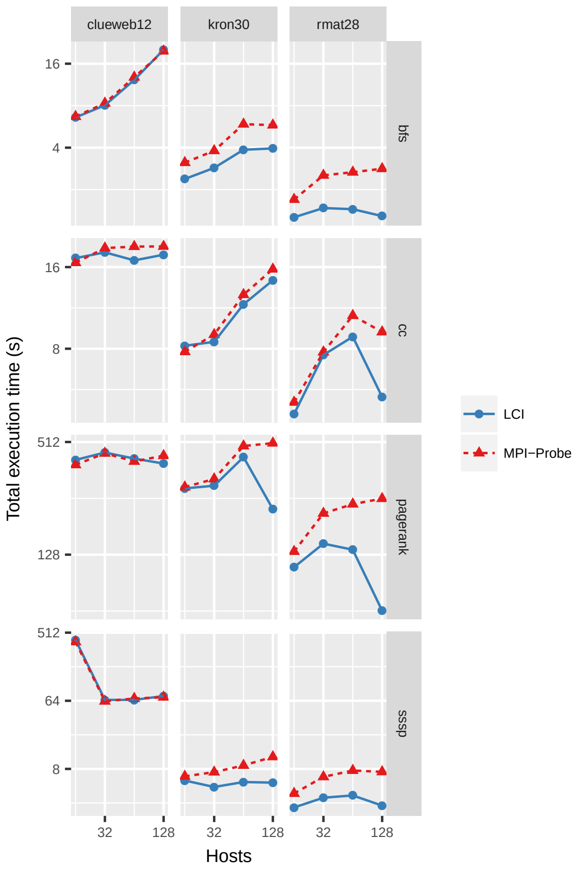


Fig. 4: Total execution time on Stampede2: Gemini with LCI and MPI-Probe runtimes.

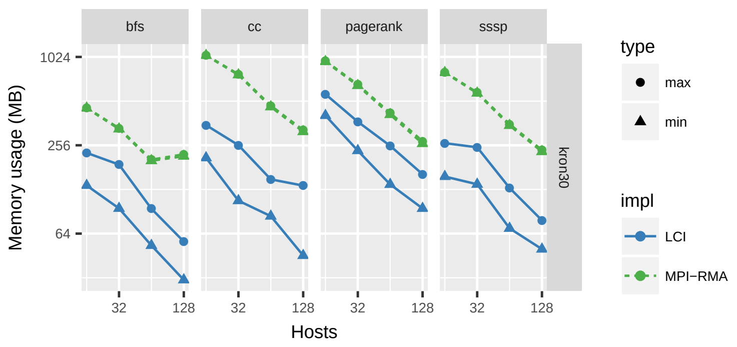


Fig. 5: Memory usage of communication buffers - maximum and minimum across hosts: Abelian with LCI and MPI-RMA runtimes on Stampede2.

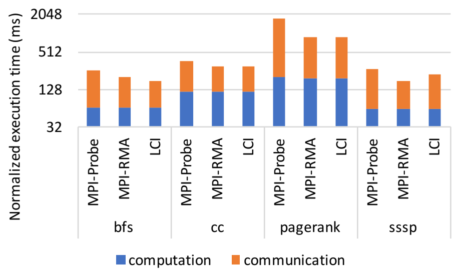


Fig. 6: Breakdown of (average) execution time of an iteration into computation and non-overlapped communication times for `kron30` at 128 hosts on Stampede2.

TABLE III: Cluster configurations.

	Stampede2	Stampede1
NIC	Omni-path	Mellanox FDR
CPU	KNL	Xeon E5
Number of cores	68	16
Clock per core	1.4 Ghz	2.7-3.5 Ghz
Memory	96GB DDR4	32GB DDR3
L3 Cache	16 GB	20 MB
MPI impl.	IntelMPI 17	MVAPICH2 2.1

TABLE IV: Total execution time (seconds) for Abelian at 128 hosts using kron30 graph with LCI and other MPI implementations on Stampede2. Timing in parentheses are window creation time which are excluded from the other results.

	bfs	cc	pagerank	sssp
LCI	**1.17**	**2.41**	**89.72**	**2.46**
IntelMPI-Probe	1.41	2.95	174.67	2.94
MVAPICH2-Probe	1.40	2.93	177.72	2.82
OpenMPI-Probe	1.33	2.99	171.57	2.82
IntelMPI-RMA (+1.4)	**1.06**	2.36	**87.84**	**1.93**
MVAPICH2-RMA (+1.8)	1.14	**2.29**	93.53	2.13
OpenMPI-RMA (+1.2)	1.21	2.34	93.74	2.25

maximum injection rate, which is typically a small constant times the number of hosts. The *allocator* can be any thread-safe memory manager; in our case, it is Abelian's allocator.

Due to its simple semantics, Queue can maintain a short matching queue at all times. Unlike MPI, ordering semantics are not required and not enforced. Instead, the SEND-ENQ returns any pending/completed request based on the order of the first packet arrival. We name this the *first-packet policy*. If needed, the user can ensure completion ordering by draining their pending requests before submitting more packets to the network or by maintaining an ordered list of pending requests and checking them in order. In particular, Abelian's communication thread maintains this order with respect to a specific incoming host.

The simple *first-packet policy* fits naturally into Abelian's communication layer since incoming data within a communication phase can be processed in any order. Further, since this is designed to match our higher layer, a thread can send a serialized message through SEND-ENQ and use RECV-DEQ for probing incoming messages. Abelian's communication layer maintains a list of incomplete requests, and can start freeing resources (for sent requests) or deserializing incoming data (for received requests) by simply checking the boolean-type status of each request.

IV. Experimental results

To evaluate the performance of the LCI communication layer on the Abelian and Gemini systems, we used a number of standard graph applications: breadth-first search (**bfs**), connected components (**cc**), single-source shortest path (**sssp**), and pagerank (**pagerank**). Table I shows the input graphs used in the experiments along with their properties; clueweb12 is one of the largest publicly available web-crawl graphs while rmat28 and kron30 are synthetically generated scale-free graphs. Abelian uses an advanced vertex-cut partitioning policy [27], whereas Gemini uses a simple blocked edge-cut partitioning policy [7] that tries to balance the assigned edges across hosts. Most of the experiments were done on the Texas Advanced Computing Center's Stampede2 KNL Cluster (Stampede2). We also perform a subset of experiments on the Stampede SandyBridge Cluster (Stampede1). The summary of each cluster is shown in Table III. All code is compiled using gcc version 7.1.0 and 4.9.3 on Stampede2 and Stampede1 respectively. In each cluster, we selected the default MPI implementation that is available; Section IV-B2 presents some results for other MPI implementations. We present the mean execution time of 5 runs using one thread per core, excluding graph construction time. All algorithms are run until convergence, except for pagerank which is run up to 100 iterations.

A. MPI-based vs. LCI communication layer

Figure 3 shows the execution time of Abelian programs with the LCI, MPI two-sided (MPI-Probe), and MPI one-sided (MPI-RMA) communication layers. With MPI two-sided, Abelian does not scale well due to the high overheads of multi-threaded communication and the irregular communication patterns that MPI_PROBE does not handle well. LCI on the other hand, is able to achieve comparable or better performance than MPI-RMA at various settings. RMA window creation time is excluded in MPI-RMA results, otherwise LCI outperforms MPI-RMA in all cases. We also observe that the improvement is more significant when the application runs with more iterations where there are more communication rounds like in the case of pagerank. The advantage of LCI vs. MPI-RMA is really in the memory usage, which is sometimes reflected in performance. At 128 hosts, LCI achieves a geometric mean speedup of $1.34\times$ over MPI-Probe and $1.08\times$ over MPI-RMA.

To determine the size of the working set of communication buffers, we instrumented the code to count the size of allocation and deallocation of the buffers. The memory usage or footprint of a host is the maximum size of the working set during execution. Figure 5 shows the maximum and the minimum memory footprints across hosts of LCI compared to MPI-RMA (*this excludes the memory used internally by MPI and only considers the allocated memory by Abelian's code*). The memory footprint of LCI is much smaller for all applications on all hosts than MPI-RMA. Due to its design, LCI can quickly recycle buffers, thus reducing memory usage and improving locality. Maximum and minimum memory footprints for MPI-RMA are close to each other. The memory usage of MPI-RMA can be up to an order of magnitude higher than that of LCI because MPI-RMA has to preallocate all buffers with a size that is the upper-bound of memory required for communication.

To confirm that the performance improvement comes from the communication layer, we analyze the kron30 results in more detail. We measured the computation time of each iteration or round on each host. We consider the maximum across hosts for each iteration and take the sum of those values to report the computation time. The rest of the execution time is the non-overlapped communication time. Figure 6 shows the time spent in computation and non-overlapped communication for `kron30` on 128 hosts. As expected, the changes in performance come from the communication component. In most applications, LCI performs best, or comparable to MPI-RMA. LCI outperforms MPI-Probe since it has lower communication overhead and outperforms MPI-RMA because of the reuse of communication buffers.

B. LCI generality

1) Other graph analytical systems: Gemini is a state-of-the-art distributed memory graph analytical framework [7] which relies on communication from many threads with `MPI_THREAD_MULTIPLE` in a similar fashion as Abelian. In particular, `MPI_PROBE` is used frequently inside a receiving thread to receive incoming messages (traversing nodes from different hosts and with different sizes). For this reason, Gemini is a good candidate to test the applicability of the LCI runtime to other frameworks. We made simple modifications to the Gemini runtime such that each sending/receiving thread uses LCI Queue instead of MPI. We evaluate the same benchmarks on the same platform, with LCI and MPI two-sided (MPI-Probe)[4].

Figure 4 presents the total execution time of Gemini with MPI-Probe and LCI. All algorithms are run until convergence. The performance behavior (or difference) is roughly similar to that of the Abelian system. In `kron30` and `rmat28` where communication overheads present a significant fraction of the total communication, we see significant improvement in performance by using LCI. Across all applications at 128 hosts, the geometric mean speedup of LCI over MPI-Probe in communication is $2\times$, yielding an execution time speedup of $1.64\times$.

Abelian and Gemini are systems with different pros and cons; comparing them is not the focus of this paper. The communication techniques described in Section III are applicable to both systems, as demonstrated. Similarly, LCI can be used as a communication runtime plug-in to improve other distributed graph analytical systems.

2) Other MPI implementations: One may argue that a better MPI implementation can improve the performance, though we believe the differences between LCI and MPI are fundamental. To verify that via empirical results, we ran some experiments using OpenMPI (commit `f9b157`) and MVAPICH 2.3b (both are latest at the time tested and configured with `psm2`) on Stampede2. The results in Table IV show that LCI remains the winner compared to other MPI implementations. There is no clear winner between different MPI implementations, though IntelMPI-RMA performs best in the majority of cases. LCI is again closest in performance to RMA implementations, and is better if we include time for window creation in the result.

[4]We did not reimplement Gemini with MPI one-sided since this requires significant changes in order to preserve computation-communication overlap.

3) Other NICs: To show that LCI and its performance is portable to other NICs, we ran a subset of experiments on the Stampede1 cluster which is equiped with a Mellanox Infiniband FDR network. We do not focus on this cluster because it has fewer cores on each host (16 compared to 68) and is an older supercomputer. Nevertheless, the results show a similar trend, LCI performs better in all tested cases and closely matches the performance in the Stampede2 cluster as shown in Table II; the exception is that MPI-RMA is actually the slowest. We believe this is because locality of communication is the bottleneck in this system, which has less cache and a slower memory subsystem than Stampede2.

V. Related work

Many frameworks for distributed-memory graph analytics have been discussed in the literature [1]–[10]. Most of these systems use either MPI or TCP/IP as the underlying communication layer.

Several communication libraries have been developed to provide lower-level support to parallel programming languages and libraries. This includes ARMCI [28], an library developed to support Global Arrays, and GASNet [29], developed to support PGAS languages such as UPC [30]. Neither of these libraries is designed to cope with high thread counts.

Other communication libraries based on the active message (AM) paradigm have been proposed as appropriate for problems with irregular, dynamic communication patterns [31], [32]. The use of AM provides great flexibility but introduces an unnecessary software overhead for many simple data-transfer patterns. Moreover, it is typically prohibited to perform blocking or time-consuming operations on an AM handler. They may also force the use of a CPU proxy for communications targeting GPUs. A possible promising direction would be to upload handlers to the NIC [33] but this raises system management issues that have plagued similar approaches in the past.

UCX [14] and Libfabric [25] are actively under development as generic communication layers; both provide great flexibility but do not optimize for a specific domain. Further, our initial investigation of these libraries does not show good performance with threads.

VI. Conclusions

To the best of our knowledge, LCI is the first communication interface targeting graph analytics that can handle high thread counts and leverage modern NIC capabilities. The design of LCI is based upon the studies of a state-of-the-art graph analytics system called Abelian, through analyzing and evaluating the performance of two existing Abelian's MPI-based communication layers. To demonstrate that LCI can be used with other graph analytics systems, we integrated it with Gemini, another state-of-the-art graph analytics framework. Our experiments show that LCI-based communication system reduces communication time by a factor of up to $2\times$ for a collection of standard graph analytics benchmarks. LCI requires only a few primitive network operations, allowing it to be easily ported to other systems. We have implemented LCI on top of `ibverbs`, `psm2`, and `Libfabric`, which is sufficient for LCI to run on almost all modern platforms.

In future work, we plan to integrate LCI more closely with the Abelian runtime to reduce the overhead of their interaction, and overlap computation and communication more effectively. Another direction of research is to port this system to heterogeneous architectures. Abelian already runs on heterogeneous platforms consisting of multicores and GPUs [34], so it should be possible to accomplish this with reasonable effort.

Acknowledgement

This research was supported by National Science Foundation (NSF) grant numbers 1337217, 1337281, 1406355, 1618425, 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used the Extreme Science and Engineering Discovery Environment [35], supported by NSF grant numbers ACI-1548562, through allocation TG-CCR130058 and TG-CIE170005.

References

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 17–30.

[2] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD Intl Conf. on Management of Data*, ser. SIGMOD '10, 2010, pp. 135–146.

[4] "Apache Giraph," http://giraph.apache.org/, 2013.

[5] A. Buluc and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *GRADES*, 2013.

[7] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI*, 2016, pp. 301–316.

[8] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 291–305.

[9] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "PGX.D: A fast distributed graph processing engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 58:1–58:12.

[10] F. Yang, M. Wu, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "GraM: Scaling graph computation to the trillions," in *SoCC*. ACM, August 2015.

[11] L. Lu, "The diameter of random massive graphs," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000, pp. 912–921.

[12] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016.

[13] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The TAO of parallelism in algorithms," in *PLDI*, 2011, pp. 12–25.

[14] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: an open source framework for HPC network APIs and beyond," in *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 2015, pp. 40–43.

[15] P. Dhabaleswar, "OSU Micro-Benchmarks 5.3," http://mvapich.cse.ohio-state.edu/benchmarks/, 2016, [Online; accessed 3-April-2017].

[16] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 2016, pp. 1–14.

[17] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hemmert, "An evaluation of MPI message rate on hybrid-core processors," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 415–424, 2014.

[18] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 30:1–30:12.

[19] H.-V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded MPI implementations," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 314–324.

[20] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP*, 2013, pp. 456–471.

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

[22] H.-V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded MPI implementations," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017.

[23] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda, "Scalable graph500 design with MPI-3 RMA," in *CLUSTER*, Sept 2014, pp. 230–238.

[24] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the MPI 3.0 one-sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[25] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the OpenFabrics interfaces-a new network API for maximizing high performance application efficiency," in *HOTI*. IEEE, 2015, pp. 34–39.

[26] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *SIGPLAN Not.*, vol. 48, no. 8, pp. 103–112, Feb. 2013.

[27] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013.

[28] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *Parallel and Distributed Processing*, pp. 533–546, 1999.

[29] D. Bonachea, "GASNet specification, v1. 1," University of California at Berkeley, Tech. Rep. CSD02-1207, 2002.

[30] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.

[31] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 401–410.

[32] M. Besta and T. Hoefler, "Accelerating irregular computations with hardware transactional memory and active messages," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 161–172.

[33] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming processing in the network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 59:1–59:16.

[34] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication optimizing framework for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. ACM, 2018.

[35] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "XSEDE: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.