LCI: a Lightweight Communication Interface for Efficient Asynchronous Multithreaded Communication

Jiakun Yan
jiakuny3@illinois.edu
University of Illinois Urbana-Champaign
Urbana, IL, USA

Marc Snir
snir@illinois.edu
University of Illinois Urbana-Champaign
Urbana, IL, USA

Abstract

The evolution of architectures, programming models, and algorithms is driving communication towards greater asynchrony and concurrency, usually in multithreaded environments. We present LCI, a communication library designed for efficient asynchronous multithreaded communication. LCI provides a concise interface that supports common point-to-point primitives and diverse completion mechanisms, along with flexible controls for incrementally fine-tuning communication resources and runtime behavior. It features a threading-efficient runtime built on atomic data structures, fine-grained non-blocking locks, and low-level network insights. We evaluate LCI on both Inifiniband and Slingshot-11 clusters with microbenchmarks and two application-level benchmarks. Experiment results show that LCI significantly outperforms existing communication libraries in various multithreaded scenarios, achieving performance that exceeds the traditional multi-process execution mode and unlocking new possibilities for emerging programming models and applications.

Keywords

Communication Library, Multithreaded Message Passing, MPI, LCI, GASNet-EX

1 Introduction

High-performance computing (HPC) architectures have become increasingly heterogeneous with extensive on-node parallelism [26, 33], while applications employ complex algorithms with sparsity or adaptivity [1, 25, 35]. In addition, new asynchronous, task-oriented programming models with runtime resource management and scheduling are becoming more popular [5, 9, 12, 28]. These trends are leading to a shift of application communication characteristics: multiple threads can logically initiate communications simultaneously; more asynchronous point-to-point communications are being used, as opposed to collective communication of bulk-synchronous styles; and there can be more simultaneously pending fine-grained communications and more opportunities for computation-communication overlap.

These characteristics fall out of the original focus of MPI, the de facto standard HPC communication library designed over 30 years ago. Since then, new communication libraries and MPI features have been introduced to tackle the asynchrony. Multiple research efforts, mainly by the MPI community, have been taken to improve multithreaded communication support. However, they still fall short of the needs of applications due to limited flexibility and constrained design space.

- **Limited Flexibility**: Each communication library only offers a limited selection of communication mechanisms. However, modern programming systems and/or applications can need combinations of many communication mechanisms. Clients often must implement their communication mechanisms on top of the existing library interface. This requires a significant effort and is not optimal when the library does not expose low-level functionality.
- **Constrained Design Space**: Most communication libraries were not designed with multithreaded performance in mind from the beginning. Existing efforts to improve multithreaded communication support (mainly for MPI) are hence handicapped by legacy code base and backward compatibility concerns, resulting in a solution that is not optimal in terms of both performance and programmability.

Suboptimal communication support, in turn, complicates the innovation of new programming models, forcing developers to adopt workarounds such as funneling communication through a single thread [49], hacking into inner communication layers [13], or using proxy processes for communication progressing [58].

To address these issues, we present the Lightweight Communication Interface (LCI), a communication library designed from scratch with asynchronous multithreaded communication in mind. It provides a unified interface that supports flexible combinations of all common point-to-point communication primitives, including send-receive, active messages, and RMA put/get (with/without notification), and various built-in mechanisms to synchronize with pending communications, including synchronizers, completion queues, function handlers, and completion graphs. In addition, the interface offers both a simple starting point for users to program and a wide range of options for them to incrementally fine-tune the communication resources and runtime behaviors, minimizing potential interference between communication and computation. Finally, it is supported by a lightweight and efficient runtime optimized for threading efficiency and massive parallelism. The runtime is built with a deep understanding of low-level network activities and employs optimizations such as atomic-based data structures, thread-local storage, and fine-grained nonblocking locks.

We evaluate LCI with microbenchmarks, a k-mer counting mini-app, and an astrophysics AMT-based application on Infiniband and Slingshot-11 clusters. The results show that LCI outperforms existing communication libraries, including standard MPI, MPICH with the VCI extension, and GASNet-EX, by a large margin in multithreaded performance while maintaining comparable single-threaded performance.

The rest of the paper is organized as follows: Section 2 discusses related works. Section 3 introduces LCI's communication interface

and shows how it can seamlessly support dynamic programming systems. Section 4 presents the key designs in LCI runtime. Section 5 analyzes the evaluation results. Section 6 concludes the paper and discusses future works.

2 Related Work

2.1 Asynchronous Communication

MPI-1 [45] was designed around coordinated communication paradigms, including two-sided send-receive and collective operations. It was developed at a time when most HPC applications followed the Bulk-Synchronous Parallel (BSP) programming model, which alternates computation and communication phases, effectively synchronizing all cores in the system. The BSP model becomes increasingly problematic as core counts increase, their compute speeds vary, and applications become more irregular. In contrast, asynchronous models allow threads to issue communication in an uncoordinated manner.

New MPI features have been proposed since then to improve support for asynchronous communication. RMA operations have been included in MPI since MPI-2, though its "window" abstraction still operates in a partially collective style. The MPI continuation proposal [43] recently introduced a way for clients to attach callbacks to pending MPI operations, aiming for more efficient polling in the case of heavy communication overlapping.

A range of communication libraries has also been proposed. GASNet[11] and the later GASNet-EX [10] focus on one-sided active messages and RMA primitives. They are intended to be used by runtime developers or as a compilation target, so their interfaces are generally more complicated than MPI. At a higher level, PGAS libraries/languages, such as UPC [20], UPC++ [7], and OpenSHMEM[14], rely on RMA operations to maintain a global address abstraction. Recently, new communication libraries have been proposed. YGM [47] features a batch-processing active messages interface and utilizes aggregation for better throughput. UNR [21] emphasizes notifiable RMA operations, optimizing them for multi-NIC aggregation and ease of use.

UCX[44] and Libfabric[41] provide low-level abstractions that are portable across multiple interconnects. They offer more flexible interfaces but at a much lower level. They also require manual bootstrap. Their primary usage is to support communication libraries rather than high-level programming systems/applications.

While these libraries have made significant progress in supporting asynchronous communication, they often provide a limited selection of features that cannot fully fulfill the communication needs of complicated runtime systems/applications. LCI improves upon these libraries by providing a more comprehensive and flexible interface that allows for a broader range of communication patterns and optimizations. It also has an additional performance focus on multithreaded communication. [52] presents an overview and some considerations of an earlier version of the LCI interface in a workshop paper.

2.2 Multithreaded Communication

All major communication libraries can be configured to be thread-safe, but the resulting performance is often suboptimal. The work on MPI and GASNet started when processors had a single core, so multithreading was not a concern. Some aspects of the interface design proved problematic when multithreading was retrofitted. Furthermore, as early applications were single-threaded, MPI implementers focused on single-threaded performance. Consequently, users kept communication single-threaded (one process per core or one communication thread per process model), reinforcing the emphasis on single-threaded performance.

Most of the existing work related to multithreaded communication optimization is based on MPI, primarily for MPICH. Assuming that serialized access to some shared resources is unavoidable, a line of work [3, 4, 8, 19, 42] studies various ways to reduce the lock contention inside MPI, including minimizing the scope of critical sections and smart lock management strategies that use priorities. Recent research has explored ways to remove the need for serialization by replicating low-level network resources. Some of it [42, 55, 56] conforms to the MPI specification by associating distinct network resources with distinct communicators and/or tags. Other research, including the endpoint proposal [17, 18, 46, 53] and the later MPICH stream proposal [57], directly add new constructs to the MPI standard, giving users direct control over network resource mappings. Similar ideas have also been adopted in OpenSHMEM [34] and GASNet-EX [27] to improve the multithreaded performance of RMA operations (but not for GASNet-EX's active message due to its progress semantics). Their approaches are relatively more direct than those proposed for MPI, as RMA operations generally do not need to bother with the progress guarantee and matching semantics. [23, 24] use message aggregation across threads to alleviate the multithreaded performance penalty. It has been included in the MPI 4.0 specification as partitioned communication.

Our work builds upon the valuable insights of existing works and advances them through completely redesigning the communication interface and runtime, free from backward compatibility concerns. By adopting appropriate interface options and semantics, decomposing the runtime into multiple independent resources, and applying various optimization techniques, we present a communication library that, for the first time known to us, achieves multithreaded performance surpassing multi-process performance at the microbenchmark level.

3 LCI Interface

The LCI interface is designed to be intuitive, flexible, and explicit, allowing LCI to be seamlessly integrated into complicated runtimes with diverse communication needs. We first present the *Objectifed Flexible Functions* (OFF) idiom that allows users to specify optional arguments in any order as a C++ function call. All LCI functions have a variant adopting this idiom. We then walk through the core LCI interface by building an LCI backend for a simple Remote Procedure Call (RPC) library. Finally, we discuss other important details of the LCI interface.

3.1 Objectified Flexible Function

The LCI interface is designed to be flexible and customizable, allowing users to express their communication needs in a straightforward and efficient manner. As a result, some LCI operations have many optional arguments. The C++ optional argument semantic is not

flexible enough to handle them, as it only allows users to specify the optional arguments in the order they are defined with no gaps.

We propose a new C++ idiom called *Objectified Flexible Function* (OFF) to overcome this restriction. It allows users to specify the optional arguments in any order, similar to the named optional arguments in Python functions. Listing 1 shows what it is like to invoke an OFF operation using the *post_send* operation in LCI as an example. Line 1 invokes the *post_send* operation in its standard form with only the positional arguments. Lines 2-3 invoke the OFF variant of the same operation. Line 2 associates the send with a specific device, and Line 3 further specifies the *rank_only* matching policy. The OFF variant in LCI is always suffixed with *_x*.

```
1 auto ret = post_send(rank, buf, size, tag, comp);
2 auto ret = post_send_x(rank, buf, size, tag, comp).device
      (device)();
3 auto ret = post_send_x(rank, buf, size, tag, comp).
      matching_policy(matching_policy_t::rank_only).device
      (device)();
```

Listing 1: Objectified Flexible Function Example.

The OFF idiom allows LCI to maintain the API's conciseness while providing as much flexibility as possible. The user can start from the simplest form and incrementally refine the communication behavior in any direction they need.

Under the hood, an OFF is implemented as a functor with a constructor that takes the positional arguments and a set of setter methods for the optional arguments. We use a Python script to generate the OFF definition based on a DSL input.

3.2 Example: LCI for iRPCLib

3.2.1 The iRPCLib Example. Remote Procedure Calls (RPCs) are a popular programming paradigm that allows a client to invoke arbitrary functions on a server. The main difference between RPC and active message is that the active message handler is executed inside the low-level communication progress engine and thus is supposed to be short with restricted functionalities (e.g., cannot invoke another communication). In contrast, RPC handlers usually have no restrictions. RPCs are used extensively in high-level programming models [7, 29, 37]. This section illustrates the LCI interface by building an LCI backend for an imaginary RPC library (iRPCLib).

```
1  // shared resources
2  lci::comp_t shandler; // send completion handler
3  lci::comp_t rcq; // receive completion queue
4  lci::rcomp_t rcomp;  // remote completion handle for rcq
5  // thread-local resources
6  __thread lci::device_t device;
7
8  // callback for source-side completion
9  void send_cb(status_t status) {
10   // free the message buffer once the send is done
11   std::free(status.buf);
12 }
13
14 void global_init(int *rank_me, int *rank_n) {
15   lci::g_runtime_init();
16   *rank_me = lci::get_rank_me();
17   *rank_n = lci::get_rank_n();
18   shandler = lci::alloc_handler(upper_layer::send_cb);
19   rcq = lci::alloc_cq();
20   rcomp = lci::register_rcomp(rcq);
21 }
22
23 void global_fina() {
24   lci::free_comp(&shandler);
25   lci::free_comp(&rcq);
26   lci::g_runtime_fina();
27 }
28
29 void thread_init() {
30   device = lci::alloc_device();
31 }
32
33 void thread_fina() {
34   lci::free_device(&device);
35 }
36
37 bool send_msg(int rank, void* buf, size_t s, int tag) {
38   lci::status_t status = lci::post_am_x(rank, buf, s,
       shandler, rcomp).tag(tag).device(device)();
39   if (status.error.is_retry())
40     return false; // the send failed temporarily
41   if (status.error.is_done())
42     send_cb(status); // the send immediately completed
43   else
44     assert(status.error.is_posted());
45   return true; // the send succeeded
46 }
47
48 // msg_t is a message descriptor type
49 // defined in the upper layer
50 bool poll_msg(msg_t *msg) {
51   lci::status_t status = lci::cq_pop(cq);
52   if (status.error.is_done()) {
53     lci::buffer_t buf = status.get_buffer();
54     *msg = {
55       .rank = status.rank,
56       .tag = status.tag,
57       .buf = buf.base,
58       .size = buf.size,
59     }
60     // the upper layer is responsible for freeing the
61     // buffer once it consumes the message
62     return true;
63   } else {
64     assert(status.error.is_retry());
65     return false;
66   }
67 }
68
69 bool do_background_work() {
70   return lci::progress_x().device(device)();
71 }
```

Listing 2: The example implementation of the iRPCLib LCI backend.

Listing 2 shows the example implementation of the iRPCLib LCI backend. We assume iRPCLib has two layers, the upper layer and the backend layer. The upper layer is responsible for registering the user-provided RPC handlers into indices and serializing and deserializing the RPC arguments into consecutive memory buffers (not shown here). The backend layer is responsible for sending the RPC handler index (*tag*) and serialized arguments (pointed by *buf*) to the target rank (*send_msg* in Line 37) and deliver the incoming messages to the upper layer (*poll_msg* in Line 50). For simplicity, we assume iRPCLib just wants the backend layer to free the message buffer once the send completes locally (*send_cb* in Line 9). We further assume iRPCLib is multithreaded. The main thread will call *global_init* (Line 14) and *global_fina* (Line 23) and all threads will call *thread_init* (Line 29) and *thread_fina* (Line 33) during the initialization and finalization phases. All threads can produce and consume communication (a.k.a. calling *send_msg* and *poll_msg*). In addition, all threads will periodically call *do_background_work*

(Line 69) to make progress on the pending communication. The backend abstraction described here is a simplified version of the HPX parcelport abstraction [51] and the Charm++ Converse Machine Interface [30].

3.2.2 Runtime Lifecycle. LCI does not have global initialization or finalization functions. Instead, it provides functions to (de)allocate a *runtime* object. The *runtime* object wraps default configurations and communication resources for LCI to operate. Most LCI operations accept *runtime* as an optional argument. In Listing 2, iRPCLib just uses the global default runtime (*g_runtime*) for simplicity (Lines 15, 26). Once at least one runtime is active, the user can query the rank of the current process (Line 16) and the total number of ranks (Line 17).

An LCI client typically allocates only one *runtime* object. However, multiple *runtime* objects can exist due to library composition. In these cases, the *runtime* abstraction enables different libraries to use different configurations and resources without interfering with each other.

3.2.3 Resource. Communications operate on resources. LCI allows users to allocate resources explicitly and associate them with communications. Resources can have a list of attributes. Users can explicitly set them during resource allocation and query them afterward. In Listing 2, iRPCLib uses one device per thread to improve threading efficiency (Line 6). A *device* encapsulating a complete set of low-level network resources and LCI ensures threads operating on different devices will not interfere with each other. In addition, iRPCLib uses a shared completion handler (*shandler* in Line 2) for source completion and a shared completion queue (*rcq* in Line 3) for target completion. Line 20 further registers the completion queue into a remote completion handle (*rcomp*) for other processes to post active messages to. (See Section 3.2.5 for more detail.)

Other important LCI resources (not shown here) include (a) *matching engines* matching send and receive; (b) *packet pools* (de)allocating fixed-sized pre-registered internal buffers (packets); and (c) *backlog queue* storing temporarily postponed communication requests. A communication operation is free to associate with any combination of these resources. For example, if the iRPCLib also uses send-receive, all threads can use a shared matching engine while using per-thread devices. In this way, it could achieve great threading efficiency while maintaining a global matching domain. Section 4.1 talks about resources in more detail.

3.2.4 Communication Posting. Line 38 uses the LCI active message operation to send the message along with a tag to the target rank using the thread-local device. LCI supports all commonly used point-to-point communication paradigms, including send/receive, active message, and RMA put/get. It supports them in a unified manner to reduce the API's complexity and allow users to easily switch between different communication models.

LCI adopts the following communication abstractions: A communication moves the data from a *source buffer* to a *target buffer*. The communication is *complete* on the source side when the source buffer can be overwritten and on the target side when the target buffer can be read. When the communication is locally complete, a *completion object* will be signaled. A *communication posting* operation submits the parameters that specify the data movement and completion signaling. A *completion checking* operation checks the completion objects for the completion status of posted requests.

The parameters needed to specify a communication are mostly the same across all point-to-point communication paradigms. Different communication paradigms are just different choices of where to specify these parameters. For example, *send-recv* specifies only the local parameters on each side, while *RMA put/get* specifies all parameters on only one side.

Therefore, LCI offers a generic communication posting operation, *post_comm*. This operation takes the target rank, the local buffer, the message size, and the local completion object as positional arguments. It takes a wide range of optional arguments, among which the most important ones include the direction, the remote buffer, and the remote completion object. Table 1 shows how combining the three optional arguments can specify the common point-to-point communication paradigms.

Direction	Remote buffer	Remote completion	Validity	Description
OUT	none	none	Yes	send
OUT	none	specified	Yes	active message
OUT	specified	none	Yes	RMA put
OUT	specified	specified	Yes	RMA put w. signal
IN	none	none	Yes	receive
IN	none	specified	No	
IN	specified	none	Yes	RMA get
IN	specified	specified	Yes	RMA get w. signal

Table 1: How *post_comm* can be used to express all common communication paradigms.

For convenience purposes, LCI also offers five derived communication operations: *post_send/recv/am/put/get*. These operations are just syntactic sugar for *post_comm* with the optional arguments set to the corresponding values.

3.2.5 Operation Return Values. An LCI communication posting operation returns a status object in one of the four categories:

- *done*: The operation has been completed immediately, and the completion objects will not be signaled.
- *posted*: The operation has been posted, and the completion objects will be signaled when the operation is complete.
- *retry*: The operation needs to be resubmitted due to temporary resource unavailability.
- *fatal error*: The operation has failed due to a fatal error.

Fatal errors are reported through C++ exceptions. The returned *status_t* object reports the other three categories. Each category includes multiple error codes to deliver more information (e.g., what resource is temporarily unavailable). When the status is *done*, the returned *status* object contains valid information about the completed operation.

Line 39-44 shows how iRPCLib handles these return values. It just returns *false* if it gets a retry error (Line 39). In this case, the upper layer can do something meaningful, such as polling other task queues or aggregating RPC messages. If the communication is immediately completed, the return *status* object will contain valid information, and iRPCLib just manually invokes *send_cb* (Line 41).

Compared to the binary return values of MPI nonblocking operations, the additional *done* and *retry* provide more informative feedback, enabling finer-grained control and potentially unlocking further optimization opportunities.

Completion Checking. Once a posted communication is completed, the completion object specified by the posting operation will be signaled with a completion descriptor (the *status_t* object). In the case of Listing 2, the *send_cb* will be automatically invoked when the send completes on the source side, and the messages will be enqueued into the *rcq* when they arrive at the target rank. Line 51 shows how iRPCLib polls *rcq* for incoming messages and decodes the *status* object. The returned *buf* is expected to be freed by the upper layer with *std::free*.

Under the hood, a completion object is a functor with a virtual *signal* method that takes a *status_t* object as an argument. Derived from it, LCI defines four built-in completion object types: *handler*, *queue*, *synchronizer*, and *graph*. *Synchronizer* is similar to MPI requests but can accept multiple signals before becoming ready. *Graph* is a more advanced completion object type similar to CUDA Graph [39] that allows users to specify a set of communication operations or user-provided functions with a partial execution order. If operation u precedes operation v in that order, then v will be started only after u completes. The local partial execution order and the ordering imposed by communication operations allow intuitive implementations of complex nonblocking collective algorithms.

3.2.6 Progress. In MPI, communication progressing happens as a side effect of certain MPI calls (typically *MPI_Test** and all blocking functions). In contrast, LCI defines an explicit *progress* function. Users can select whether progress is invoked by a distinct thread or as a side-effect of other operations and how frequently progress should be called. Line 70 shows how the backend layer uses the OFF version of the progress function to make progress on the thread-local device.

3.3 Other Details

3.3.1 Other Advanced Features. Listing 2 assumes the upper layer supplies plain send buffers, and LCI also uses plain buffers to deliver incoming active messages. Alternatively, advanced users can explicitly ask LCI for packets and directly assemble the message in it. They can also instruct LCI to deliver incoming active messages in packets. These practices can save memory copy for buffer-copy protocol.

In addition, LCI follows the common practice of many low-level communication libraries by providing an explicit memory registration function. Memory registration is optional for local buffers but mandatory for remote buffers. LCI supports on-demand paging when the underlying hardware allows it.

Besides a single source and target buffer, LCI also supports transmitting a list of source and target buffers in a single communication posting operation to reduce the overheads related to request posting, handshakes, and completion signaling.

3.3.2 Send-Receive Semantics. LCI adopts the send-receive semantics proposed in [16], namely, out-of-order delivery and restricted wildcard matching, to avoid sequential bottlenecks inside the runtime. The in-order delivery and wildcard matching have long been seen as a stumbling block for efficient multithreaded MPI implementation, as they require centralized matching queues that are hard to parallelize. Weakening them allows LCI to adopt a more efficient hashtable-based matching engine. By default, LCI matches send and receive by the (matching engine, source rank, tag) tuple on the target side. Users can still achieve in-order matching for send-receives by encoding ordering information into the tag field. They can also set the *matching_policy* optional argument to *tag_only* or *rank_only* when posting sends and receives to achieve wildcards similar to those of MPI, except that the sender needs to know the sent message will be matched by a wildcard receive call. Under the hood, the *matching_policy* will instruct the matching engine on how to make the insertion key based on *rank* and *tag*. Users can also achieve more flexible matching policies by supplying their own *make_key* function.

4 LCI Runtime

Communication activities inside the LCI runtime are carefully decomposed into operations of multiple independent resources, while each resource is carefully optimized with threading efficiency in mind. Key optimizations include atomic-based data structures, fine-grained locking, thread-local storage, and try lock wrappers.

4.1 LCI Resources

4.1.1 Prerequisite: Multi-Producer-Multi-Consumer (MPMC) Array. We find it a common need for LCI to store certain resources in an array for future reference. Such arrays are rarely written but frequently read, and the array size is usually unknown at compilation time. For example, the registered completion object array is only written (appended) during a new registration (usually not on the critical path) but is read whenever an active message or RMA with notification message is received. We do not want to preallocate a large array as it may waste memory and restrict the total number of registered completion objects.

To meet this need, we implement a simple MPMC array that supports dynamic resizing and fast read. It borrows the key idea in [2]: a *write* and *append* (and the potential *resize*) is protected by a lock to prevent missed writes, but *read* is lock-free. Every *resize* swaps the old array with a new one that doubles the size, and the deallocation of the old array is postponed to prevent the *read* from reading invalid memory.

4.1.2 Packet Pool. The packet pool is responsible for efficient allocation (get) and deallocation (put) of fixed-sized pre-registered buffers, which we call *packets*. *get* can be nonblocking and will return a *nullptr* when it fails the first packet stealing attempts (and *post_comm* returns *retry*). The packet pool is implemented as a collection of thread-local double-ended queues (deque). The list of thread-local deques is managed by an MPMC array. By default, every thread puts and gets packets from its own deque. When the local deque is empty, the thread will try stealing half of the total packets from a randomly selected deque. Local packet put and get are performed at the tail end, and packet stealing is performed at the head end to achieve better cache locality. Thread safety is achieved with a per-deque spinlock, so there should be no thread contention during normal operation.

4.1.3 Matching Engine. The matching engine is responsible for matching the incoming sends with user-posted receives at the target side. It contains two major methods: *make_key* generates a matching key based on *source rank*, *tag*, and user-supplied *matching_policy*; *insert* tries inserting a key-value pair with a *type* (send or receive) and will either return 0, meaning the entry has been inserted, or the matched values if an entry with the same key and a complementary type has been found. The default implementation is based on a hashtable where each bucket is a list of queues. Thread safety is achieved with a per-bucket spinlock, and we do not expect severe thread contention, given that the bucket number (by default 65536) is significantly larger than the thread number (on the order of tens to hundreds). Special optimization is applied to the case when a bucket contains no more than three queues and when a queue contains no more than two sends or receives, where we use fixed-size arrays instead of linked lists for the buckets/queues. Therefore, when the load factor is low, the hashtable can perform an insertion with a single cache miss.

4.1.4 Completion Objects. All LCI built-in completion objects are atomic-based. *Synchronizer* is implemented as an atomic flag (when expecting one signal) or a fixed-sized array protected by two atomic counters (when expecting multiple signals). *Completion queue* has two implementations: one based on the state-of-the-art LCRQ [38] and the other based on a hand-written Fetch-And-Add-based fixed-sized array. *Completion handler* is essentially a function and does not need any special treatment. Every node in the completion graph uses an atomic counter to track the number of received signals. Every ready node will be immediately fired, and a completed node will signal all its descendants.

4.1.5 Backlog Queue. The backlog queue is used to store communication requests that cannot be immediately submitted and cannot be back-propagated to the user. For example, when the progress engine wants to send a handshake message, but the underlying network send queue is full. LCI expects such scenarios to be rare, so we implement it with a simple C++ queue with a spinlock. An atomic flag prevents the progress engine from unnecessarily polling an empty backlog queue.

4.2 Network Backend

4.2.1 The Network Backend Layer. LCI isolates different network backends from its core runtime with a simple network backend wrapper. The backend abstraction operates on two resources: network context and network device. Each LCI runtime maps to a network context, which contains global network resources. Each LCI device maps to a network device, which contains network resources accessed on the critical path.

All communication operations on the critical path are posted to a network device. These operations include posting network-layer send/recv/write/read, polling for completed operations, and (de)registering memory. LCI does not require the ability to handle tag matching and unexpected receive from the network backends. The LCI progress engine ensures there are always enough pre-posted receives in the device. LCI expects two threads operating on different network devices not to interfere with each other.

Currently, LCI supports two full-fledged network backends: *libibverbs* (ibv) [40] and *libfabric* (ofi) [41].

4.2.2 Trylock Wrapper. Lower-level network stacks such as ibv and ofi generally use spin locks to ensure thread safety and usually blockingly acquire them. To mitigate the cost of blocking on these locks, we examine the backend source code to identify the lock granularity and wrap all corresponding accesses with a try lock. For example, an ibv completion queue is protected by a spin lock, so we create a spin lock for each ibv completion queue at LCI layer and *try_lock* the corresponding LCI-layer lock before we access the ibv completion queue through (*ibv_poll_cq*). If the try lock fails, we will return the *retry* error code to the caller. This gives LCI clients more optimization opportunities during network contention.

4.2.3 libibverbs Analysis. libibverbs is the lowest-level public API for Infiniband. It can also be run on top of High-speed Ethernet devices through RDMA over Converged Ethernet (RoCE). We focus on its *mlx5* provider here as it is the latest and most widely used one. Each *libibverbs* queue pair, shared receive queue, and completion queues are protected by their own spinlock. In addition, each queue pair is associated with a set of hardware resources (micro User Access Region or uUAR) that are protected by its own lock on the host side. Different queue pairs may share the same uUAR [54]. *libibverbs* users can use *thread domains* to explicitly associate queue pairs with uUARs. The memory (de)registration functions do not acquire any locks.

The LCI ibv backend puts an ibv completion queue, an ibv shared receive queue, and a collection of ibv queue pairs in a network device. LCI uses a try lock wrapper for every ibv completion queue and shared receive queue. An LCI device attribute *ibv_td_strategy* controls the way LCI uses thread domains. By default, it will create a thread domain for every ibv queue pair (the *per_qp* strategy). Users can also ask LCI to allocate a single thread domain for all queue pairs of a device (the *all_qp* strategy) or not use thread domains at all (the *none* strategy). The *all_qp* strategy is recommended when each thread has a dedicated LCI device. LCI uses a try lock wrapper for every queue pair in the *per_qp* case and uses a try lock wrapper for all queue pairs of the device in the other two cases.

With libibverbs, LCI can provide a contention-free guarantee not only for threads operating on different devices but also for threads operating on different ibv data structures (queue pairs, completion queues, shared receive queues). This means there will be no interference between a worker thread posting communication and a background thread progressing the communication, which is typical in asynchronous programming systems such as AMTs [9].

4.2.4 libfabric Analysis. libfabric is a portable low-level network API that supports many network providers. It is also currently the lowest-level public API for HPE Slingshot-11. The LCI ofi backend is designed with the libfabric cxi provider and verbs provider in mind. Both providers have similar lock granularity: every endpoint has a single spin lock; all *post_send/recv* on the endpoint and *poll_cq* on associated completion queues need to acquire the endpoint lock; the memory (de)registration function involves the use of a registration cache, which is allocated per domain and is protected with a pthread mutex.

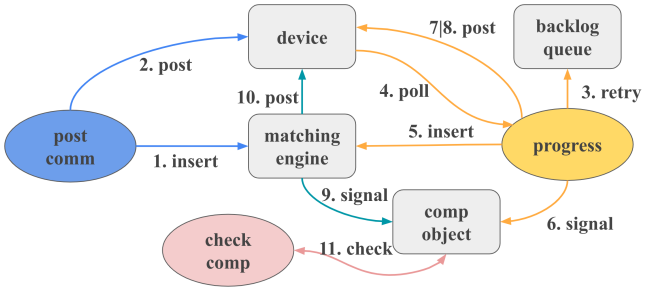


Figure 1: LCI Runtime Architecture. Operations are represented as circles and resources as rectangles. The packet pool is omitted for clarity.

The LCI ofi backend, therefore, puts in a network device an ofi domain, endpoint, and completion queue. It uses a single try lock wrapper for the endpoint. It does not employ a try lock wrapper for the memory (de)registration functions, as LCI does not yet have a way to backpropagate the memory registration failure to the user.

In general, the current libfabric provider implementation has a coarser lock granularity than libibverbs, which makes it less efficient in multi-threading scenarios. However, the libfabric interface is general enough to accommodate additional optimizations in the provider implementation. libfabric defines a more advanced *FI_THREAD_FID* threading support level that only requires serialization to individual libfabric objects. Combined with libfabric's scalable endpoint, it could achieve lock granularity similar to that of libibverbs. However, current providers do not exploit this threading support level with additional optimizations. The two providers also do not support the scalable endpoint feature.

4.3 Communication Protocol

LCI adopts communication protocols similar to existing communication libraries, so we will briefly mention them due to page limit. For the send-receive and active message operations, depending on the message size, LCI adopts three different communication protocols: inject, buffer-copy, and zero-copy. For put/get operations, LCI directly translates them into the corresponding low-level network operations. Due to the lack of support for *RDMA read with notification* in the interconnects we have access to, LCI does not implement the *get with signal* communication operation for the time being.

4.4 Putting Everything Together

Figure 1 shows an overview of the LCI runtime architecture. When the user posts a communication, (1) if it is a receive, a receive descriptor will be inserted into the matching engine; (2) otherwise, the communication request will be posted to the device. When the user invokes the progress engine, it will (3) first check the backlog queue and retry the communication requests in that queue; and (4) poll the device for completed operation and react accordingly. The reaction may involve (5) inserting an incoming send into the matching table, (6) signaling a completion object, (7) replenishing the pre-posted receives, or (8) posting another communication request to the device as part of the rendezvous (zero-copy) protocol. When either the communication posting procedure or the progress engine finds a match in the matching engine, it will either (9) signal the completion object or (10) post another communication to continue the rendezvous protocol. (11) The completion checking procedure will query the completion object for the status of posted communication.

For simplicity, the figure omits the packet pool. The packet pool can be involved in (2, 7, 8, 10) when either the user or the progress function tries to post communication requests to the device. In addition, the communication request could be pushed into the backlog queue in (2) if the user disallows the retry return value and in (7, 8, 10) as the progress engine cannot keep retrying the communication requests.

4.5 Implementation Note

LCI is implemented as a C++11 library with the CMake build system. It is also available as a Spack package. LCI supports four bootstrapping backends: PMI1, PMI2, PMIx, and MPI. It has been tested on Infiniband, RoCE, Slingshot-11, and Ethernet networks. It is fully open-sourced with the MIT license.

5 Evaluation

5.1 Experimental Setup

We evaluate LCI on SDSC Expanse and NCSA Delta. Table 2 shows their configuration. Expanse uses InfiniBand, which is one of the most widely used interconnects for HPC clusters and accounts for 61% of the Top500 systems. Delta uses Slingshot-11, which is increasingly popular and used on 7 of the top-10 systems.[1] All experiments are conducted at least six times. The figures show the average and standard deviation.

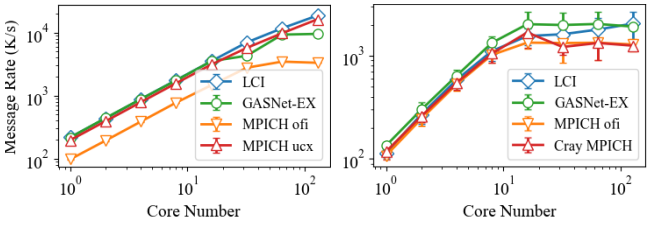
Table 2: Platform Configuration.

Platform	SDSC Expanse	NCSA Delta
CPU	AMD EPYC 7742	AMD EPYC 7763
sockets/node	2	2
cores/socket	64	64
NIC	Mellanox ConnectX-6	HPE Cassini
Network	HDR InfiniBand (2x50Gbps)	Slingshot-11 (200Gbps)
Software	MPICH 4.3.0	MPICH 4.3.0
	GASNet 2025.2.0	GASNet 2025.2.0
	UCX 1.17.0	Cray MPICH 8.1.27
	Libfabric 1.21.0	Libfabric 1.15.2.0
	Libibverbs 43.0	

5.2 Micro-benchmarks

In asynchronous multithreaded applications, message rate and bandwidth are more critical than latency due to communication overlapping and nonblocking execution. Therefore, we use these two metrics to compare LCI with standard MPI, MPICH with the VCI extension, and GASNet-EX. Our micro-benchmarks run on two nodes with two basic modes. The *process-based* mode uses one process on each core, while the *thread-based* setting uses one process

[1] Statistics are based on the TOP500 List published in Nov. 2024.



(a) On Expanse (InfiniBand). (b) On Delta (Slingshot-11).

Figure 2: Process-based message rate micro-benchmark. We use one process per core and one thread per process.

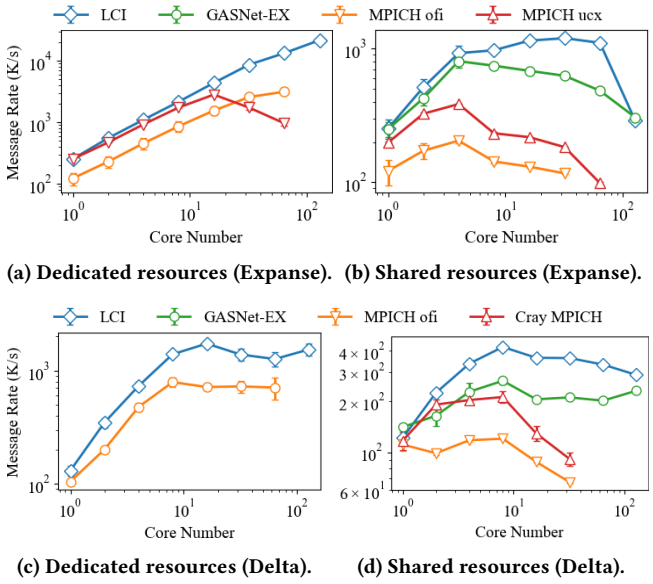
on each node with one thread per core. Each process/thread has a peer process/thread on the other node, and it performs ping-pongs with the peer. Existing multithreaded applications can either share a global set of communication resources or, if the application logic and underlying communication library permit, allocate dedicated resources for each thread. Therefore, the thread-based mode is further divided into two sub-modes according to the resource-sharing pattern: (a) in the *dedicated resource* mode, each thread allocates its communication resources; (b) In the *shared resource* mode, all threads share a global set of communication resources. The dedicated resource mode is implemented with MPICH VCIs and LCI devices. Cray-MPICH and GASNet-EX do not support this mode. We also set *mpi_assert_no_any_tag* and *mpi_assert_allow_overtaking* to *true* and configure *MPIR_CVAR_CH4_GLOBAL_PROGRESS* to 0 to minimize the thread contention on VCIs.

To ensure uniformity across different communication libraries, we build a simple layer (the Lightweight Communication Wrapper, or LCW) on top of LCI, MPI, and GASNet-EX and use it to write the microbenchmarks. The microbenchmarks, along with the LCW layer, are open-sourced[2]. LCW implements simple nonblocking active messages and send-receive primitives. For MPI, it uses *MPI_Isend*/*MPI_Irecv* for send-receive and *MPI_Isend*/pre-posted *MPI_Irecv* for active messages. For GASNet-EX, it uses *gex_AM_RequestMedium* for active messages and does not support send-receive due to implementation complexity. We show the active message results in the message rate microbenchmark and the send-receive results in the bandwidth microbenchmark.

5.2.1 Single-threaded Performance. Figure 2 shows the single-threaded message rate results. We fixed the message size to 8 bytes and increased the process number from 1 to 128 per node. Each process/thread runs 100k iterations. We report the uni-directional message rate. LCI achieves performance comparable to the other communication libraries. Figures for the single-threaded bandwidth results are omitted due to page limit, but the results are similar.

5.2.2 Multithreaded Performance. Figure 3 shows the multithreaded message rate results. We fixed the message size to 8 bytes and increased the thread number from 1 to 128 per node. LCI achieves significant speedups in multithreaded performance on both platforms (sometimes more than 10x). In particular, multithreaded LCI with dedicated devices achieves even slightly better performance than multi-process LCI (around 15% at full scale). The MPICH's

[2]https://github.com/<anonymous>/lcw



(a) Dedicated resources (Expanse). (b) Shared resources (Expanse).

(c) Dedicated resources (Delta). (d) Shared resources (Delta).

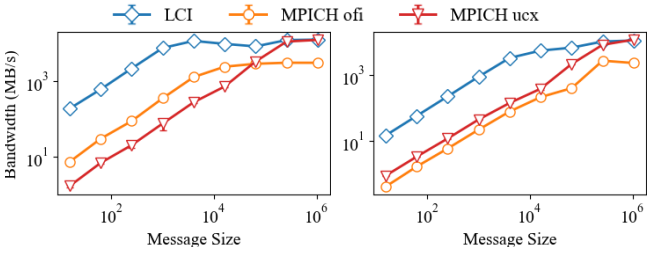
Figure 3: Thread-based message rate micro-benchmark. We use one process per node and one thread per core. *Dedicated resources* uses one LCI device/MPICH VCI per thread. *Shared resources* uses one set of resources for the entire process.

VCI extension greatly helps multithreaded performance, but the overall performance is still suboptimal. GASNet-EX shows good multithreaded performance in the shared resource mode, but its lack of resource-replication support weakens its competencies if the application wants to use more resources.

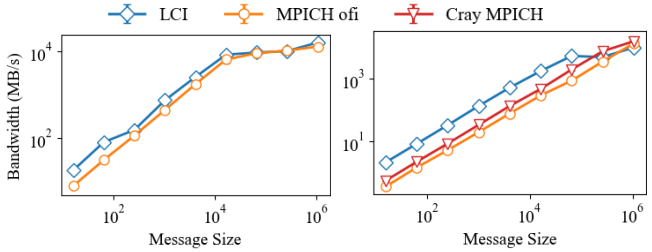
Even though we do not directly evaluate UCX and Libfabric due to the difficulty of bootstrapping and the complexity of their APIs, the MPICH results on Expanse (particularly Figure 3a) give hints on their multithreaded performance. UCX is generally faster than libfabric on InfiniBand, but its performance degrades sharply when there are more than 16 threads. Libfabric shows good scaling results with dedicated resources at the cost of absolute performance numbers. LCI achieves the best of both worlds by directly building on the lowest-level public API, libibverbs. UCX does not support Slingshot-11, so its results on Delta are unavailable. MPICH does not support more than 64 VCIs, so some data points are missing.

Figure 4 shows the multithreaded bandwidth results for various message sizes. We fix the thread number to 64 to avoid inter-socket overheads. We increase the message size from 16B to 1 MiB. Each process/thread runs 1k iterations. We report the unidirectional bandwidth. Similar to the message rate results, LCI also achieves significant speedup in multithreaded bandwidth. GASNet-EX is absent here due to its lack of send-receive support.

5.2.3 Individual Resources. LCI communications involve operations on a variety of resources. Each resource is optimized for threading efficiency, and users can explicitly allocate multiple replicas of them. Our next microbenchmark evaluates the threading efficiency of three major LCI resources: completion queue, matching engine, and packet pool. All microbenchmarks run on a single node on Delta with different thread numbers. All threads perform



(a) Dedicated resources (Expanse). (b) Shared resources (Expanse).



(c) Dedicated resources (Delta). (d) Shared resources (Delta).

Figure 4: Thread-based bandwidth micro-benchmark. We use one process per node and one thread per core. *Dedicated resources* uses one LCI device/MPICH VCI per thread. *Shared resources* uses one set of resources for the entire process.

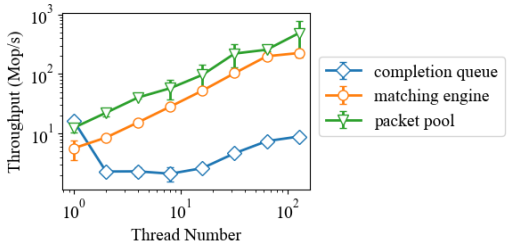
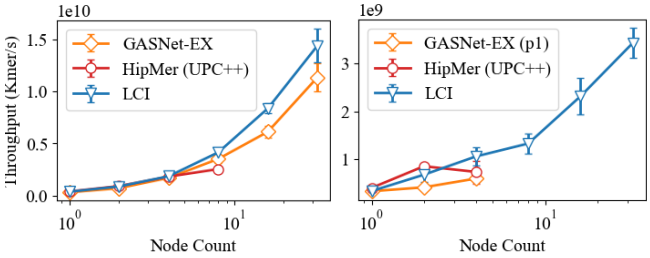


Figure 5: Maximum throughput of individual resources over different thread numbers.

100k of key resource methods that are used in the communication critical path (a pair of completion queue push/pop, matching engine inserts, or packet pool get/put). Figure 5 shows the results. As we can see, the packet pool and matching engine scales well with thread number, achieving 480 Mops (Million operations per second) or 225 Mops with 128 threads. As a reference, our ping-pong microbenchmark achieves at most 22 Million Messages per second (Figure 3a). This means allocating one instance of each resource per process is sufficient. However, the completion queue only achieves 9 Mops with 128 threads, which means applications aiming for higher throughput may need to allocate more completion queues per process. The completion queue throughput is primarily constrained by how fast threads can perform the atomic fetch-and-add operation on a shared variable. Our message rate microbenchmark shown above uses one completion queue per thread.



(a) Expanse (with InfiniBand) (b) Delta (with Slingshot-11)

Figure 6: K-mer counting strong scaling results comparing multithreaded LCI, GASNet-EX, and single-threaded UPC++ (HipMer reference implementation). *GASNet-EX (p1)* means dedicating one thread for network progress.

5.3 K-mer Counting

Our first application-level benchmark is k-mer counting, an important step in bioinformatics for analyzing biological sequences. The mini-app used here is based on the version used in the de novo genome assembler HipMer[22]. With error-prone reads of DNA sequences as its input, the k-mer counting mini-app computes the histogram of the number of occurrences of k-mers. A read is a DNA sequence that is shorter than the actual DNA strand, while a k-mer is a short DNA sequence of a fixed size k.

In the k-mer counting stage, HipMer traverses the dataset twice. The first traversal inserts the k-mers into a two-layer Bloom filter. A Bloom filter is a space-efficient data structure that tests whether an element is in a set with a small false positive rate. The second traversal then consults the Bloom filter and inserts those with more than one occurrence into a hashmap. The hashmap maintains the actual count of the k-mers, while the two-layer Bloom filter is used to reduce the memory footprint of the hashtable by filtering out those occurring only once (which are likely erroneous).

HipMer is written in UPC++ with only one thread per process. Each k-mer is statically mapped to a process using a hash function. Each process reads part of the dataset and sends the k-mers to the mapped processes via UPC++ RPCs. It further employs an aggregation buffer per target process to reduce communication overhead.

We implement a multithreaded version of the HipMer k-mer counting stage. The new implementation is also based on the RPC abstraction and aggregation, with libcuckoo hashtable[32] and a hand-written atomic-based Bloom filter. It supports two network backends, LCI and GASNet-EX, primarily leveraging their active message primitives. The LCI backend shares many similarities with the one described in Section 3.2. Compared to the single-threaded implementation, multithreading reduces the number of aggregation targets by a factor of N, where N is the thread number per process. All threads can serve the incoming RPCs, resulting in improved load balance.

We run the k-mer counting mini-app with the human chr14 dataset (7.75GB). It contains 37 million reads and 1.8 billion k-mers (with the k-mer length $k = 51$). We run the multithreaded implementation with 2 processes per node to avoid the inter-socket

overheads. The aggregation buffer size is set to be 8KB per destination. The total aggregation buffer size is always smaller than its HipMer counterpart due to the reduced destination number. All threads run the application logic and periodically progress the network backend (the *all-worker setup*). This is the best setup for LCI on both platforms. However, when running GASNet-EX on Delta, the all-worker setup results in devastating performance (over 20x worse than LCI). Therefore, we add an additional *dedicated progress setup* for GASNet-EX: use 63 threads for application logic and one thread for network progress. We report the better of the two setups for GASNet-EX (all-worker setup on Expanse and dedicated progress setup on Delta).

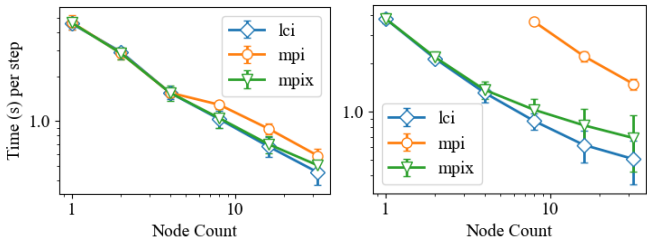
Fig.6 shows the strong scaling results of the mini-app on Expanse and Delta from 1 node (2 processes/128 cores) to 32 nodes (64 processes/4096 cores). Our multithreaded implementation outperforms the single-threaded reference implementation by up to 60% on Expanse (8 nodes) and 40% on Delta (4 nodes), at which point the reference implementation suffers from severe load imbalance problems across 512/1024 processes. In addition, the LCI backend outperforms its GASNet-EX counterpart by 35% on Expanse (16 nodes) and 75% on Delta (4 nodes). Although not shown here, we also tried larger aggregation buffer sizes (up to 64KB), which resulted in slightly smaller gaps between GASNet-EX and LCI due to less frequent communication but lower overall performance due to worse load balance.

HipMer evaluation stops at 8/4 nodes because UPC++ takes too long to bootstrap for larger process counts. Investigation shows it was due to the slow PMI2 fence operation. Multithreaded GASNet-EX on Delta beyond 4 nodes will run into deadlock. We are working with the GASNet-EX team to investigate this issue.

5.4 HPX and Octo-Tiger

The increasingly complicated architectures and dynamic scientific computing algorithms have attracted growing interest in the Asynchronous Many-Task (AMT) programming model [5, 6, 9, 29, 31]. With AMTs, users express their application logic as a set of fine-grained tasks and task dependencies. The runtime then schedules these tasks on available resources according to task dependencies and data locality. Compared to the traditional bulk-synchronous parallel (BSP) model, AMTs can potentially achieve better load balance, portability, and communication overlapping, with lower user programming complexity. However, previous works have shown that existing communication libraries usually do not support AMTs' communication needs most efficiently, as their communications are heavily multithreaded and asynchronous [13, 49, 51].

[50] has integrated a previous C version of LCI into HPX [29], an established AMT runtime that fully complies with the C++ Standard APIs and extends them to the distributed case. In this work, we upgrade the LCI support inside HPX to the latest C++ version and evaluate its performance with an astrophysics application, Octo-Tiger [35], which simulates the evolution of stellar systems based on adaptive octo-trees and fast multipole methods. Octo-Tiger is built on top of HPX for fully asynchronous execution and communication overlapping. We use the "rotating star" scenarios and report the timestamp per step.



(a) Expanse (with InfiniBand) **(b) Delta (with Slingshot-11)**

Figure 7: Octo-Tiger strong scaling results comparing LCI, standard MPI, and MPICH with the VCI extension (mpix).

Figure 7 shows the results. For *mpix*, we use the MPICH VCI extension. Preliminary experiments have shown that MPICH with VCI extensions performs better with the libfabric backend than the UCX backend. Therefore, we use the MPICH libfabric backend here. Results reported here use the optimal VCI number for *mpix* and the optimal device number for *lci*. We also use replicated request pools for *mpix* to reduce thread contention on completion polling. On Expanse, LCI outperforms *mpi* (standard MPI) by 30% and *mpix* by 10%. On Delta, LCI outperforms *mpi* by 3x and *mpix* by 35%. In addition, *mpix* needs 8 VCIs on both platforms to reach the optimal performance, while *lci* only needs 1 device on Expanse and 2 devices on Delta. This shows that LCI has better intra-resource threading efficiency compared to MPICH, thanks to its thread-efficient runtime design.

Related work. A previous version of LCI has been integrated into PaRSEC and showed favorable performance over its MPI backend [36]. LCI has also been integrated into HPX releases and used by other projects. [48] implements a 2D FFT mini-app with HPX and reports that the LCI backend outperforms the MPI counterpart and the reference FFTW implementation by 5x. [15] scales Octo-Tiger to 1700+ GPU nodes (around 7000 GPUs/processes) on Perlmutter and achieves 1.7x speedup compared to Cray MPICH at full scale. We are unable to conduct similar scaling experiments in this paper due to computation resource limitations, but we expect the latest version of LCI to have similar scalability.

6 Conclusion and Future Work

We have presented LCI, a communication library designed for asynchronous multithreaded programming models and applications. LCI is designed to be flexible, easy to use, explicit, and efficient. It has shown significant performance improvements over existing communication libraries such as MPICH and GASNet-EX in micro-benchmarks and applications.

LCI is still under active development. There are several areas for future improvements. We list some of the most important ones below.

Collective Communication: The existing LCI API focuses on point-to-point communication primitives, as they are more commonly seen in asynchronous multithreaded applications and are also the basic building blocks for collective communication. LCI offers a few

basic collective communication primitives, including dissemination-based barrier and tree-based broadcast/reduce. Users can also combine LCI with other communication libraries, such as MPI or *CCL, for better collective communication performance. Identifying the right interface and efficient design of non-bulk-synchronous collective communication for asynchronous multithreaded applications is an important future work.

GPU Communication: The existing LCI focuses on CPU-CPU communication, as it is the most common use case in asynchronous multithreaded applications. However, we recognize that GPU-direct communication is becoming increasingly popular and important for LCI's future work. There are mature techniques for integrating GPU-Direct RDMA into existing communication libraries, such as MPI and GASNet-EX, so we do not expect it to be difficult for LCI. GPU-initiated communication is a different issue, and we are still looking for potential applications and use cases.

References

[1] Sameh Abdulah, Allison H. Baker, George Bosilca, Qinglei Cao, Stefano Castruccio, Marc G. Genton, David E. Keyes, Zubair Khalid, Hatem Ltaief, Yan Song, Georgiy L. Stenchikov, and Ying Sun. 2024. Boosting Earth System Model Outputs And Saving PetaBytes in Their Storage Using Exascale Climate Emulators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 2, 12 pages. doi:10.1109/SC41406.2024.00008

[2] Andrei Alexandrescu and Maged M. Michael. 2004. Lock-Free Data Structures with Hazard Pointers. https://erdani.org/publications/cuj-2004-12.pdf

[3] Abdelhalim Amer, Huiwei Lu, Pavan Balaji, Milind Chabbi, Yanjie Wei, Jeff Hammond, and Satoshi Matsuoka. 2019-01-08. Lock Contention Management in Multithreaded MPI. *ACM Transactions on Parallel Computing* 5, 3 (2019-01-08), 12:1–12:21. doi:10.1145/3275443

[4] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015-01-24. MPI+Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA) *(PPoPP 2015)*. Association for Computing Machinery, 239–248. doi:10.1145/2688500.2688522

[5] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the Gap Between CUDA and Task Parallelism. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17. doi:10.1109/SC41406.2024.00049

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing* (Berlin, Heidelberg) *(Lecture Notes in Computer Science)*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.). Springer, 863–874. doi:10.1007/978-3-642-03869-3_80

[7] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973. doi:10.1109/IPDPS.2019.00104

[8] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2008. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg) *(Lecture Notes in Computer Science)*, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer, 120–129. doi:10.1007/978-3-540-87475-1_20

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012-11. Legion: Expressing Locality and Independence with Logical Regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. doi:10.1109/SC.2012.71

[10] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Languages and Compilers for Parallel Computing: 31st International Workshop (LCPC 2018)*. Springer, 138–158. doi:10.1007/978-3-030-34627-0_11

[11] Dan Bonachea and Jaein Jeong. 2002. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring* 31 (2002), 17.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. 2013-11. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013-11), 36–45. doi:10.1109/MCSE.2013.98

[13] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moreto, Martin Schulz, Ramon Beivide, Mateo Valero, and Abhinav Bhatele. 2019. Optimizing computation-communication overlap in asynchronous task-based programs. In *Proceedings of the ACM International Conference on Supercomputing*. 380–391.

[14] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, New York, USA) *(PGAS '10)*. Association for Computing Machinery, New York, NY, USA, Article 2, 3 pages. doi:10.1145/2020373.2020375

[15] Gregor Daiß, Patrick Diehl, Jiakun Yan, John K Holmen, Rahulkumar Gayatri, Christoph Junghans, Alexander Straub, Jeff R Hammond, Dominic Marcello, Miwako Tsuji, et al. 2024. Asynchronous-Many-Task Systems: Challenges and Opportunities–Scaling an AMR Astrophysics Code on Exascale machines using Kokkos and HPX. *arXiv preprint arXiv:2412.15518* (2024).

[16] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards millions of communicating threads. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2966884.2966914

[17] E.D. Demaine, I. Foster, C. Kesselman, and M. Snir. 2001. Generalized Communicators in the Message Passing Interface. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (2001), 610–616. doi:10.1109/71.932714

[18] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting* (New York, NY, USA) *(EuroMPI '13)*. Association for Computing Machinery, 13–18. doi:10.1145/2488551.2488553

[19] Gábor Dózsa, Sameer Kumar, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Joe Ratterman, and Rajeev Thakur. 2010. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Recent Advances in the Message Passing Interface* (Berlin, Heidelberg) *(Lecture Notes in Computer Science)*, Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra (Eds.). Springer, 11–20. doi:10.1007/978-3-642-15646-5_2

[20] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (Tampa, Florida) *(SC '06)*. Association for Computing Machinery, New York, NY, USA, 27–es. doi:10.1145/1188455.1188483

[21] Guangnan Feng, Jiabin Xie, Dezun Dong, and Yutong Lu. 2024. UNR: Unified Notifiable RMA Library for HPC. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[22] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages. doi:10.1145/2807591.2807664

[23] Ryan Grant, Anthony Skjellum, and Purushotham V. Bangalore. 2015. *Lightweight Threading with MPI Using Persistent Communications Semantics*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States). https://www.osti.gov/servlets/purl/1328651

[24] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Vol. 11501. Springer International Publishing, 330–350. doi:10.1007/978-3-030-20656-7-17

[25] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex C Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydın Buluç, et al. 2020. Terabase-scale metagenome coassembly with MetaHipMer. *Scientific reports* 10, 1 (2020), 10689.

[26] HPCwire. 2024. Venado: The AI Supercomputer Built to Tackle Science's Biggest Challenges. https://www.hpcwire.com/2024/09/16/venado-the-ai-supercomputer-built-to-tackle-sciences-biggest-challenges/

[27] Khaled Z. Ibrahim and Katherine Yelick. 2014. On the Conditions for Efficient Interoperability with Threads: An Experience with PGAS Languages Using Cray Communication Domains. In *Proceedings of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 23–32. doi:10.1145/2597652.2597657

[28] Hartmut Kaiser et al. 2020. HPX - The C++ Standard Library for Parallelism and Concurrency. *Journal of Open Source Software* 5, 53 (2020), 2352.

[29] Hartmut Kaiser et al. 2023. STEllAR-GROUP/hpx: HPX V1.9.0: The C++ Standard Library for Parallelism and Concurrency. doi:10.5281/zenodo.598202

[30] Laxmikant V Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Josh Yelon. 1996. Converse: An interoperable framework for parallel programming. In *Proceedings of International Conference on Parallel Processing*. IEEE, 212–217.

[31] Laxmikant V. Kale and Sanjeev Krishnan. 1993-10-01. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *ACM SIGPLAN Notices* 28, 10 (1993-10-01), 91–108. doi:10.1145/167962.165874

[32] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. doi:10.1145/2592798.2592820

[33] LLNL. [n. d.]. Lawrence Livermore National Laboratory's El Capitan verified as world's fastest supercomputer. https://www.llnl.gov/article/52061/lawrence-livermore-national-laboratorys-el-capitan-verified-worlds-fastest-supercomputer

[34] Wenbin Lu, Tony Curtis, and Barbara Chapman. 2019. Enabling Low-Overhead Communication in Multi-threaded OpenSHMEM Applications using Contexts. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. 47–57. doi:10.1109/PAW-ATM49560.2019.00010

[35] Dominic C Marcello, Sagiv Shiber, Orsola De Marco, Juhan Frank, Geoffrey C Clayton, Patrick M Motl, Patrick Diehl, and Hartmut Kaiser. 2021. Octo-Tiger: a new, 3D hydrodynamic code for stellar mergers that uses HPX parallelization. *Monthly Notices of the Royal Astronomical Society* 504, 4 (2021), 5345–5382.

[36] Omri Mor, George Bosilca, and Marc Snir. 2023. Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine. In *Proceedings of the 52nd International Conference on Parallel Processing* (New York, NY, USA) *(ICPP '23)*. Association for Computing Machinery, 153–162. doi:10.1145/3605573.3605642

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[38] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 103–112. doi:10.1145/2442516.2442527

[39] NVIDIA. 2019. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/

[40] NVIDIA. 2025. RDMA Aware Networks Programming User Manual. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17

[41] OFI Working Group (OFIWG). 2024. Libfabric Programmer's Manual.

[42] Thananon Patinyasakdikul, David Eberius, George Bosilca, and Nathan Hjelm. 2019. Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. doi:10.1109/CLUSTER.2019.8891015

[43] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. 2021-09-01. Callback-Based Completion Notification Using MPI Continuations. *Parallel Comput.* 106 (2021-09-01), 102793. doi:10.1016/j.parco.2021.102793

[44] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 40–43. doi:10.1109/HOTI.2015.13

[45] Marc Snir. 1998. *MPI–the Complete Reference: the MPI core*. Vol. 1. MIT press.

[46] Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar. 2014. Enabling Efficient Multithreaded MPI Communication through a Library-Based Implementation of MPI Endpoints. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 487–498. doi:10.1109/SC.2014.45

[47] Trevor Steil, Tahsin Reza, Benjamin Priest, and Roger Pearce. 2023. Embracing irregular parallelism in hpc with ygm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[48] Alexander Strack, Christopher Taylor, Patrick Diehl, and Dirk Pflüger. 2024. Experiences Porting Shared and Distributed Applications to Asynchronous Tasks: A Multidimensional FFT Case-Study. In *Workshop on Asynchronous Many-Task Systems and Applications*. Springer, 111–122.

[49] Philippe Swartvagher. 2022. *On the Interactions between HPC Task-based Runtime Systems and Communication Libraries*. Ph. D. Dissertation. Université de Bordeaux.

[50] Jiakun Yan, Hartmut Kaiser, and Marc Snir. 2023. Design and Analysis of the Network Software Stack of an Asynchronous Many-task System – The LCI parcelport of HPX. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1151–1161. doi:10.1145/3624062.3624598

[51] Jiakun Yan, Hartmut Kaiser, and Marc Snir. 2025. Understanding the Communication Needs of Asynchronous Many-Task Systems–A Case Study of HPX+ LCI. *arXiv preprint arXiv:2503.12774* (2025).

[52] Jiakun Yan and Marc Snir. 2025. Contemplating a Lightweight Communication Interface for Asynchronous Many-Task Systems. *arXiv preprint arXiv:2503.15400* (2025).

[53] Rohit Zambre and Aparna Chandramowlishwaran. 2022. Lessons Learned on MPI+threads Communication. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, 1–16.

[54] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. 2018. Scalable Communication Endpoints for MPI+Threads Applications. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 803–812. https://ieeexplore.ieee.org/abstract/document/8645059

[55] Rohit Zambre, Aparna Chandramowliswharan, and Pavan Balaji. 2020. How I Learned to Stop Worrying about User-Visible Endpoints and Love MPI. In *Proceedings of the 34th ACM International Conference on Supercomputing* (New York, NY, USA) *(ICS '20)*. Association for Computing Machinery, 1–13. doi:10.1145/3392717.3392773

[56] Rohit Zambre, Damodar Sahasrabudhe, Hui Zhou, Martin Berzins, Aparna Chandramowlishwaran, and Pavan Balaji. 2021. Logically Parallel Communication for Fast MPI+Threads Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 3038–3052. doi:10.1109/TPDS.2021.3075157

[57] Hui Zhou, Ken Raffenetti, Yanfei Guo, and Rajeev Thakur. 2022. MPIX Stream: An Explicit Solution to Hybrid MPI+X Programming. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA '22)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3555819.3555820

[58] Xingyu Zhu, Dan Huang, and Yutong Lu. 2023. Enhancing Distributed Graph Matching Algorithm with MPI RMA based Active Messages. In *2023 9th International Conference on Computer and Communications (ICCC)*. IEEE, 1952–1961.