LCI: A Lightweight Communication Interface v1.7

Marc Snir, Hoang-Vu Dang, Omri Mor, Jiakun Yan

July 14, 2023

Contents

A few of the functions described in this document are not yet implemented. They are indicated by grayed out text.

1 Introduction

The *Lightweight Communication Interface* (LCI) is a research communication library designed to be a communication layer by libraries and frameworks. It focuses on the communication needs of applications with irregular, dynamic communication patterns, and heavy multithreading (hundreds of threads per process). In particular, it is used to study the needs of asynchronous many-tasks runtimes and graph analysis frameworks. The design of LCI was influenced by our understanding of the needs of graph analytics frameworks, such as D-Galois and D-Ligra, Gemini or Powergraph [5, 12, 6] and of Asynchronous Many-Task (AMT) frameworks, such as Charm++, Habanero-C/C++, HPX, Legion and PaRSEC [9, 1, 8, 2, 3]. It was also influenced by the functionality provided by InfiniBand verb interface [10], and emerging new libraries, such as Libfabric and UCX [7, 11].

The main design goals for LCI are listed below:

1. Add to to the communication library functionality that can be directly supported by current or future Network Interface Controllers (NICs), so as to reduce communication latency. For example, LCI provides simple signaling mechanisms to indicate the completion of communication, such as setting a flag. Direct support of such mechanisms by the NIC is easy and can avoid significant polling overheads.

2. Do not add to LCI any functionality that is more efficiently supported by higher-level frameworks. For example, LCI does not provide the equivalent of MPI datatypes. Frameworks most often use a limited number of data structures and can provide efficient serialization/deserialization routines for those. Packing before sending and unpacking after receiving is almost always more efficient than using a datatype, as the first option uses compiled code, while the second uses an interpreter.

3. Directly support the communication paradigms that are needed by the frameworks we target, so as to avoid an additional layer that maps framework communication paradigms to the underlying library. For example, many frameworks poll for communication atop MPI, while MPI polls for packets; we wish to avoid these multiple levels of polling.

4. Provides more control to the framework on the allocation of compute resources for communication. Current communication libraries require an asynchronous polling agent to handle incoming messages or various

handshakes. Often, this asynchronous activities are performed as a side-effect of communication calls. As a result, communication calls can have widely varying execution time. The framework is better placed to know when more resources should be allocated to communication or computation. This issue plagues libraries using the active message paradigm.

5. Provide more control to the framework on the allocation of memory. Current libraries, such as MPI, use static allocation of memory for communication buffering. This is detrimental to applications where communication volume and topology varies widely, as is the case for graph analytics and AMTs. Support efficiently large numbers of concurrent threads and of concurrent communications.

6. Reduce the number of layers in the communication stack without losing too much programmability.

7. Facilitate the splitting of communication into independent streams. Communication rates on future systems may necessitate the use multiple communication servers; performance will suffer if they need to interact. Systems with multiple NICs are becoming more widespread and NICs can support multiple independent channels.

1.1 Communication Paradigms

The basic communication paradigm we focus on is that of *producer-consumer*: Data is moved from the producer's address space to the consumer's address space. The move can proceed asynchronously. In the general case the producer is signaled when the transfer has completed at the source and consumer is signaled when the transfer has completed at the destination. Different variants can reduce the communication overheads by taking advantage of implicit synchronization provided by the logic of the application.

1.2 Two-Sided Communication

The most general mechanism for such producer-consumer communication is a *two-sided send-receive*. On the producer side (send):

1. The producer process passes to LCI the produced data, by value (if short) or by reference (if long). It specifies a destination and a communication tag.

2. LCI ships the data and either returns the buffer to the producer or frees it.

On the consumer side:

1. The consumer passes to LCI a buffer to receive the communicated data; alternatively, the buffer may be allocated by LCI. The consumer specifies a source and a tag.

2. LCI returns the filled buffer to the consumer.

3. If the buffer was allocated by LCI, then the consumer returns it to LCI once it has consumed the buffer's content.

4. Alternatively, short data is passed via the completion mechanism, e.g., as an argument to an active message handler.

The matching of producer send to consumer receive can done by tag and rank matching, or by tag only. A message sent can be matched at the consumer side by only one of these two mechanisms, thus avoiding the performance problems encountered when "wildcard" receives are mixed with regular ones.

A communication is *complete* at the producer side when the send buffer can be reused, if LCI returns it to the application; or when the application has relinquished control of the buffer. It is *complete* on the consumer side when the receive buffer contains the received data and can be accessed by the consumer.

Some communication calls, on the producer side, are *elemental*: the communication call is locally complete when the call returns. Elemental calls return within a bounded amount of time, irrespective of the state of LCI.

Other communication calls are *split*: the call may return before the communication is locally complete; a separate mechanism is needed to signal local completion.

☞ Elemental calls are different from MPI *blocking* calls. Blocking MPI calls may take an arbitrary amount of time, depending on the timing of other MPI calls; e.g., an MPI receive may return only after a matching send occurred. The calling htread is often descheduled, and rescheduled when the call can complete. An LCI elemental call with return after a bounded amount of time, irrespective of other communications; the calling thread is not descheduled.

LCI provides a variety of mechanisms to signal the local completion of a split communication. This can take the form of a synchronization operation. If the synchronization construct is supported by the thread scheduler, the operation may directly mark a blocked thread as ready, re-enabling it for execution, thus avoiding the need for additional polling. This is useful when incoming data can be associated up-front with the thread that will consume the data.

Another mechanism is the invocation of a callback function. The callback function can be passed the communication data (by value or reference) and/or communication metadata[1].

Finally, a completion queue can be used to indicate completion and, on the consumer side, to hold the arriving message value or a reference to it, and metadata. This is useful for messages that are consumed by a runtime polling agent.

Alternatively, LCI may not provide any completion signal, in which case a "fence" operation by the application can be used to complete pending communications.

1.3 Communication Buffers

LCI provides four mechanisms for communicating data: Short messages can be passed by value to/from LCI. We call this mode *short* or *immediate*.

Medium sized messages are passed to/from LCI in a fixed-size buffer; the buffer size is typically chosen to fit into one communication packet. We call this mode *medium* or *buffered*. On the producer side, the application passes the buffer address to LCI; for fixed-size (medium) buffer, LCI has two possible choices: (1) It can copy the user buffer into an internal buffer and return, at which point the operation is locally complete; (2) it can take control of the buffer and return, at which point the operation is locally complete. On the consumer side, either the application passes the receive buffer to LCI, or the buffer is allocated by LCI. In the later case, the application has to return the buffer to LCI once it has consumed the incoming data.

Long messages are passed to/from LCI in contiguous buffers of arbitrary size; they are typically transferred via RDMA. We call this mode *long* or *direct*, although messages can be of any size. Long buffers must be in registered memory before they can be used by an RDMA operation. They can be registered up-front, or registered on the fly by LCI. On the consumer

[1]A callback function can be used to implement any other completion mechanisms. We "special-case" simple synchronization operations and completion queues, as we expect those to be implemented more directly and more efficiently by the communication stack.

side, long buffers can be specified by the application or allocated dynamically by LCI.

IO-vectors aggregate in one message multiple long buffers and one medium buffer.

Allocation by LCI may save one copy; it is often preferable when communication is irregular, so that the size of the communicated data changes rapidly; or when data needs to be gathered before being sent and scattered after being received, forcing the use of intermediary buffers.

1.4 One-Sided Communication

In one-sided communication LCI is invoked by only one of the two involved parties: Either the producer (*put*) or the consumer (*get*) provides all the information needed for the communication to occur. Ideally, the communication will be completed with no involvement of software on the other side. The different mechanisms discussed in the previous section can be used for specifying buffers (short, medium or long; allocated by application or LCI) and for signaling completion (synchronization operation, callback, completion queue). Completion can be signaled both at the local node that invokes the operation and at the remote node that is passive.

2 Design Principles

The design of LCI follows the following principles

1. LCI is a library with C bindings.

2. All calls are non blocking. To the extent possible, their execution time does not depend on other concurrent activities and communication state. Calls that cannot complete in a fixed time (such as a receive) are split into two parts: the operation start call and an operation complete signal.

3. Communication calls may return failure, so as to exercise back-pressure on the calling framework, when buffers are unavailable.

4. LCI interacts with three other systems:

   - A driver for the underlying communication hardware. Our current implementations use Libibverbs – for Infiniband devices, and Libfabric – for other generic devices.
   - Memory manager: malloc() and variants; if there are different types of memories, then allocators for each memory type.
   - Thread or task manager. LCI assumes that threads (or tasks) are preemptable and that mechanisms exist for a thread to wait on a synchronization event and for the thread to be marked ready when the event occurs[2]. LCI defines a few synchronization operations. These can be implemented atop existing synchronization objects, e.g., semaphores and condition variable, for POSIX; and similar synchronization operations for task libraries. Our current implementation interfaces with the POSIX pthread library.

   LCI needs to be built for a specific low-level communication driver and task scheduler, since those do not usually provide standard interfaces.

5. LCI does not check for invalid arguments; it trusts the higher-level framework to invoke LCI correctly[3].

6. LCI operations are invoked in two manners:

[2]It is possible to replace preemption with busy waiting, but the design focuses on the case where a waiting thread is descheduled.

[3]It should be possible to have a debug version of the LCI library with error checks.

Synchronously when the application[4] requires an LCI service

Asynchronously when an LCI action is triggered by the arrival of a message.

Asynchronous actions are executed by the NIC or by a polling agent. For synchronous invocations, the required action is determined by the called function name and its arguments; for asynchronous actions, it is determined by the content of the arriving message and local information.

7. LCI calls are designed to avoid, to the extent possible, complex control. The logic to be executed by a synchronous invocation depends on the function name; the logic to be executed in an asynchronous invocation depends on an "endpoint id" in the arriving message. In many cases, such logic can be ported to the NIC.

8. LCI calls are thread-safe. However, performance can often be improved by ensuring that objects are not contested and specialized, more efficient versions are specified in some cases when thread safety is not required.

The LCI library specifies several classes of constructs:

Objects: Many objects used by LCI are structures with an explicitly defined type. Other objects are *opaque*, with an implementation-defined type. Opaque objects are used when different implementation may need different structures. When this is not necessary, LCI defines object layout explicitly, so that framework can access their content with no function calls.

Datatypes: Datatype identifiers have the form `LCI_`*`name`*`_t`.

Constants: Constant identifier have the form `LCI_`*`NAME`* – all capital. Whenever possible, the document specifies their actual value.

Functions: Function identifiers have the form `LCI_`*`name`*

All definitions appear in an `LCI.h` header file.

[4]We use the term "application" for the code that invokes LCI. We expect "applications" to usually be libraries or frameworks.

3 Basic Concepts

3.1 Memory and data objects

LCI requires that some LCI objects be located in *registered memory* – memory that is registered with a device and can be accessed by the device. LCI provides mechanisms for registering memory segments for a device.

LCI communicates with the application by reading and writing various data objects, or invoking synchronization methods. Such objects are allocated by the application, using LCI-provided constructors or allocators; they are often required to be in registered memory. At any point in time such an object can be exclusively accessed by LCI or exclusively accessed by the application. For example, for send-receive communication, a receive buffer is passed to LCI by the receive call and returned to the application when the receive call completes. Alternatively, it can be allocated by LCI when the communication occurs and returned to LCI by the application once the application consumed the incoming message.

☞ An application may starve the LCI library by not returning LCI objects to the library in a timely manner. This may cause communications to fail for lack of resources.

3.2 Process and Rank

A parallel computation involves a fixed number of processes, numbered 0 to $size - 1$. Process rank does not change during the computation.

☞ This is unlike MPI where processes can have different ranks in different communicators. We assume that the "relative-rank" to "absolute-rank" translation is done at an application layer above LCI; this translation could be dynamic, in order to support migration.

3.3 Device

A *device* is a physical or logical resource that can be used for communication. Communications that use the same device share a resource and may affect each other's performance. Polling, if needed, is done at the granularity of a device. The device can perform RDMA operation one memory that is *registered* with the device. process "owns" at least one device, and sometimes more.

3.4 Endpoint

Communication operations target *endpoints.* Point-to-point communications involve matching endpoints on the two communicating processes A device may support multiple endpoints. An endpoint consists of a pair of *ports*: A *command port* that handles LCI commands submitted by local threads; and a *message port* than handles incoming messages. Endpoints are "application-facing". An endpoint create method specifies:

1. The device that the endpoint is associated with.
2. Some properties of the communications using this endpoint.

Endpoints at distinct processes are matched in the order of their creation. Endpoints are immutable: The logic to be executed by the command port upon an LCI call depends only on the function called, its arguments and the immutable endpoint state; the logic to be executed by the message port upon message arrival depends only on the load carried by the message and the immutable endpoint state.

Implementation Note: The lower-level protocol is likely to be stateful, as it needs to track packet arrivals and handle error recovery.

3.5 Communication

3.5.1 Point-to-Point

A basic point-to-point communication involves two matching endpoints at distinct processes and results in data being moved from a *source buffer* at the *producer process* to a *destination buffer* at the *consumer process.* The communication involves a *source endpoint* at the producer and a *destination endpoint* at the consumer.

3.5.2 Communication Arguments

For a communication to occur, one needs to specify up to five things:

Source buffer: where the data comes from

Destination buffer: where the data goes to

Source completion mechanism: how the producer process is informed that the source buffer was read and can be reused

Destination completion mechanism: how the consumer process is informed that the destination buffer was written and data is ready for consumption

Tag: additional information on how to handle messages.

The arguments are encoded in three ways:

1. The name of the invoked LCI function(s)
2. The arguments passed to the function(s)
3. The properties of the endpoints involved at both ends.

The design of LCI aims at reducing the control logic that depends on call arguments. The generic design we adopted is that the method used to signal completion of a communication, and the type of communication buffer used is a property of the endpoints; the actual addresses of communication buffers and synchronization objects may be indicated by arguments passed to the LCI call(s).

3.5.3 One-Sided vs. Two-Sided Communication

In two-sided communication (send-receive), the producer (sender) specifies the source communication parameters not determined by the source endpoint, and the consumer (receiver) specifies the destination communication parameters not determined by the destination endpoint.

In one-sided communication, communication is effected by one call, either on the producer side (put, accumulate) or the consumer side (get); this call specifies all the communication parameters not associated with the two endpoints.

In two-sided communication, send and receive are matched by tag or by tag and sender rank. A *matching table* is used to support this matching. Multiple endpoints on the same device may share the same matching table, or use separate ones. Matching tables are not shared across devices.

Implementation Note: The matching table can be implemented as one hash table storing both receives posted ahead of matching message arrival and messages arriving ahead of the matching receive [4]. There is no need for separate queues holding expected and unexpected messages. If matching involves only tag, then the sender and receiver rank is set to a special value

☞ Wildcard receives can be emulated using only tag matching. The main difference with MPI is that the send operation determines which matching mechanism will be used by the recipient; this must be matched by a suitable receive operation. The two types of matching are not mixed in LCI, avoiding some performance issues of MPI.

2-sided communication provides "duplex" synchronization: Communication occurs logically only after both producer and consumer have indicated they are ready by calling LCI. (The implementation may start communication earlier, but this is not visible to the user.) 1-sided communication has only "half-duplex" synchronization: Communication occurs as soon as one side has indicated it is ready: producer if a put is used; or consumer, if a get is used. Additional synchronization is needed to ensure that the other side is ready. One-sided communication has a performance advantage when this additional synchronization is part of the program logic and does not add significant overhead. This can be either because one synchronization serves for multiple communications; or because communications occur in both directions and a communication from a to b also indicates that a is ready to receive from b.

3.5.4 Communication Modes

LCI supports three communication modes: they differ in the size of the buffers they handle, the protocol used for communication and the management of communication buffers.

Short: A *short* message is passed to LCI by value on the producer side; on the consumer side, it is either passed by value to a handler, stored in a completion queue, or stored in a synchronizer. The size of oa short message data is implementation-dependent, but is at least 8 bytes. It will typically consist or one or few cache lines.

A short message communication does not return an error due to resource exception. Short message communication uses an *eager* protocol, where data is sent from source to destination as soon as it is available at the source.

Implementation Note: Short is currently set at 128B.

Implementation Note: It is theoretically possible to crash LCI by repeatedly sending immediate messages with no matching receives; the LCI implementation allocates enough buffer space to avoid such occurrence but for pathological cases.

TBD: Should the size be implementation-dependent, or fixed? What should be the minimum size? Can we use vector registers to speed up their handling?

Medium: A *medium* message is stored in a fixed size buffer both on the source and at the destination. The buffer is passed to LCI by reference. On the producer side, LCI can copy the data into an internal buffer, at which point the communication is locally complete; or it can take control of the buffer, avoiding the copy operation. In this later case, the buffer is deallocated by LCI (i.e., returned to the internal LCI buffer allocator), requiring no completion signaling. The last mode supports a "shoot and forget" mechanism, where communications may be initiated and continued concurrently with computation, with no further coordination with LCI.

An incoming buffered message is stored into an LCI buffer on the consumer side. It can then either be copied into a buffer provided by the consumer, or passed directly by reference to the consumer. In the later case, the consumer has to eventually return the buffer to LCI. The application is responsible for returning buffers to LCI on the consumer side in a timely manner, as buffered communications may run out of buffer space, otherwise.

An LCI buffer may be requested on the producer side either explicitly by the producer or implicitly, if the communication operation involves copying data into an LCI buffer. In both cases, an error message will be returned if LCI is out of storage. This provides a mechanism for applying back pressure to a producer that runs ahead of the consumer(s). LCI guarantees that once the data is buffered by LCI at the producer, it will be eventually transferred to the consumer.

Buffered communication will typically use an eager protocol, with data sent directly to the destination when it is available at the source. The communication protocol must ensure that lack of buffer space at the destination will not result in a failure, e.g., by using credit based flow control.

The maximum size of LCI fixed-size buffers is implementation dependent, but is at least 4K Bytes.

Implementation Note: Medium length is currently set at 8K.

TBD: We might want the buffer size to be slightly less than 4K (or 8k) so that data+metadata fits in 4K (8k)

Long: A *long* messages can be of arbitrary size. Long buffers are allocated by by the application; the destination buffer of a put can also be allocated by LCI.

On the consumer side, the incoming data is buffered either into a buffer allocated by LCI or a buffer in the application memory. In the former case, the application has to eventually return the buffer to LCI.

Direct communication mode will typically involve zero copying, as data is transferred directly from the source buffer to the destination buffer by RDMA. This may require an additional handshake in a rendezvous protocol: The producer sends a "request to send"; the consumer returns an "OK to send" message once the destination buffer is available; the producer then transfer the data to the consumer using RDMA.

A direct communication will fail if dynamic buffer allocation is required at the destination, and the destination has run out of buffer memory. In such a case, the consume returns a negative acknowledgment, and an error value will be returned at the source using the completion mechanism.

If a call requires memory allocation by LCI, the the call may fail and return an error value if memory is not available. This includes an explicit call to LCI to allocate a buffer or a buffered send that requires data to be copied into an LCI buffer.

LCI may also need to allocate destination buffers asynchronously upon message arrival. If the allocation fails, then the failure needs to be reported back to the put or send call that generated the message. The alternative is a fatal exception that is not associated with any specific LCI call. This alternative is to be avoided as much as possible. For buffered messages, it is up to the LCI implementation to ensure that such an asynchronous allocation succeeds. The flow control protocol must ensure that once a medium sized message is buffered at the source, it will be eventually transferred and buffered at the destination. Back pressure may be exercised on the producer process by failing its buffer allocation requests. This leaves the case of direct messages, when the destination buffer is allocated by LCI. LCI uses in this case a rendezvous protocol, in order to allocate the destination buffer before data is copied from the source. Allocation failure will cause a failure indication to be returned at the source by the completion event.

3.5.5 Completion

Completion of a split communication at source or at destination can be specified in several ways.

Completion queue: (CQ) When a communication completes, an entry with information on this communication is appended to the completion queue.The entry is deleted when accessed by the application. This mechanism is preferred when one communication agent handles communications on behalf of multiple workers. An application thread may poll the completion queue, or wait (block) until the queue is nonempty. The completion queue entries on the destination side include the message tag, origin endpoint and, optionally, a buffer descriptor: a value, for immediate communication, and an address and length for buffered or direct communication. A completion queue may be shared across multiple endpoints, but not across devices.

Synchronizer: (Sync) This is a synchronization object that indicates the completion of one or a fixed number of communications. A thread can wait (block) until the specified number of communications completed, or can test the synchronizer They can provide efficient support when incoming messages can be directly handled by their consumer.

☞ The meaning of "waking-up" a task or thread will depend on the task or thread package that LCI interfaces with. Most typically, the awoken task does not start executing immediately, but is only marked as ready and scheduled for execution at a later time. In a non-preemptive task model, a task stops running if it completes, yields, or blocks. When a task completes, the scheduler is invoked to start or restart another ready task.

Handler: (Handler) The call specifies a handler to execute upon completion. The handler is passed message metadata (tag, source rank or destination rank) and the data (value, or address and length). The handler may not execute LCI calls.

☞ Handlers can be used to implement completion queues or synchronizers – but direct LCI support is expected to be more efficient.

Null: No completion operation occurs. This is useful when completion can be guaranteed by other means (e.g., a fence).

3.5.6 Summary

The different point-to-point communication calls are illustrated in Table 1. `s`,`m` or `l` indicates communication mode (short, medium or long), `a` (allocated) indicates LCI allocated buffer, and `n` (nocopy) indicates that the medium buffer sent or received is owned by LCI. Some functions are not yet implemented and are grayed out.

Table 1: Possible point-to-point communications.

Call at Source	**protocol**	**Local buffer**	**Local completion**	**Matching call at destination**	**Remote buffer**	**Remote completion**
Sendl	Long	App buffer	Null, Sync, CQ, Handler	**Recvl**	App buffer	n/a

Call at Source	**protocol**	**Local buffer**	**Local completion**	**Matching call at destination**	**Remote buffer**	**Remote completion**
Sendm	Medium	App buffer	n/a	**Recvm**, **Recvmn**	LCI or App buffer	n/a
Sendmn	Medium	LCI buffer	n/a	**Recvm**, **Recvmn**	LCI or App buffer	n/a
Recvm	Medium	App buffer	Null, Sync, CQ, Handler	**Sendm**, **Sendmn**	LCI or App buffer	n/a
Recvmn	Medium	LCI buffer	Sync, CQ, Handler	**Sendm**, **Sendmn**	LCI or App buffer	n/a
Sends	Short	value	n/a	**Recvs**	Passed to CQ, Sync, or Handler	Sync, CQ, Handler
Sendv	IO vector	App buffer	Null, Sync, CQ, Handler	**Recvv**	App buffer	n/a
Recvv	IO vector	App buffer	Null, Sync, CQ, Handler	**Sendv**	App buffer	n/a
Putl	Long	App buffer	Null, Sync, CQ, handler	n/a	LCI or App buffer	NULL, CQ, Sync, Handler
Putla	Long	LCI buffer	Null, Sync, CQ, handler	n/a	LCI buffer	CQ, Sync, Handler
Getl	Long	App buffer	Null, Sync, CQ, handler	n/a	App buffer	NULL, CQ, Sync, Handler
Putm	Medium	App buffer	n/a	n/a	App buffer	Null, CQ, Sync, Handler
Putma	Medium	App buffer	n/a	n/a	LCI buffer	CQ, Sync, Handler
Putmn	Medium	LCI buffer	n/a	n/a	App buffer	Null, CQ, Sync, Handler

Call at Source	**protocol**	**Local buffer**	**Local completion**	**Matching call at destination**	**Remote buffer**	**Remote completion**
Putmna	Medium	LCI buffer	n/a	n/a	LCI buffer	CQ, Sync, Handler
Puts	Short	value	n/a	n/a	Passed to CQ, Sync, or handler	CQ, Sync, Handler
Putv	IO Vector	App buffer	Null, CQ, Sync, Handler	n/a	App buffer	CQ, Sync, Handler, Null
Putva	IO Vector	App buffer	Null, CQ, Sync, Handler	n/a	LCI buffer	CQ, Sync, Handler

3.5.7 Bulk Completion

A *fence* operations for a local endpoint completes all communications issued locally on this endpoint. (A `put` is locally-issued on the source endpoint; a `get` is locally-issued on the destination endpoint.)

3.5.8 Ordering

LCI communications are not ordered: Two messages sent by the same source to the same destination arrive out of order.

3.6 Progress

Depending on the capabilities of the NIC, some LCI communications may require the involvement of a polling software agent, in order to complete. Polling may be required at the destination, to handle incoming messages, and may be required at the source, in order to execute handshakes in a rendezvous protocol. Different communication libraries use different polling mechanisms: Polling can be executed by a dedicated progress thread; or it may be executed by computation threads, as a “side-effect” of the execution of library calls. Each choice has downsides: A dedicated progress thread may be idle most of the time; if it is descheduled by the thread scheduler, communication performance is affected. Polling as a side effect results in variable execution time for library calls; communication by one thread may affect the performance of another thread. The choice of polling mechanism may also affect cache-to-cache traffic, when one thread polls on behalf of another.

LCI leaves control of polling to the application: Progress calls can be used to make progress for a particular device; the call also specifies how long the call will poll the device before returning.

4 API

☞ The API is undergoing change. This document may not be up-to-date.

4.1 Errors

All LCI functions return an error value. Error values are of type `LCI_error_t` defined as

```
typedef enum {
  LCI_OK = 0,
  LCI_ERR_RETRY,
  LCI_ERR_FATAL,
} LCI_error_t;
```

A value of zero indicates success while a nonzero value indicates failure. Different values are used to indicate the types of failure.

4.2 Initialization and completion

The execution of LCI calls must be preceded by a call to initialize LCI, except `LCI_initialized`.

```
    LCI_error_t LCI_initialize();
```

The LCI structures are freed by the following call

```
    LCI_error_t LCI_finalize();
```

```
    LCI_error_t LCI_initialized(int *flag);
```

This call can be called before LCI is initialized or after. It return `flag = true` if LCI has been initialized, `flag = false`, otherwise.

4.3 Execution Environment

Several parameters determine the configuration of LCI. They come in two categories

Preset: Parameters that are determined by the implementation code or set at job submission time. These parameters can be queried but not updated

Environment-dependent: Parameters that are set at LCI initialization, using environment variables of the same name. If the corresponding environment variable is not set, then a predefined, implementation-dependent default value is used. The values of these parameters do not change during execution.

We describe most of these parameters in the following sections. Additional parameters control the Infiniband configuration.

`LCI_DEFAULT_TABLE_LENGTH`: initial number of entries in a matching table used to match sends and receives.

`LCI_MAX_TABLE_LENGTH`: maximum number of entries in a matching table.

4.4 Processes, Devices and Endpoints

A parallel job consists of a fixed set of communicating processes. `LCI_NUM_PROCESSES` holds the total number of processes. Each process has a fixed rank that is stored in `LCI_RANK`.

4.4.1 Device

is a physical or logical resource that can be used by a process for communication. Communications that use the same device share a resource and may affect each other's performance. Resources such as packet pools, matching tables, completion objects, and registered memory region are associated with devices and shared between endpoints of the same device. LCI initialization creates on each process a initial device `LCI_UR_DEVICE`. `LCI_NUM_DEVICES` stores the number of devices at the local process.

A device can be created by the following call

```
void LCI_device_init(LCI_device_t *device_ptr);
```

The call returns a pointer to the newly created device. This call is a collective operation. All processes are expected to call this function when they want to create a device. It will result in a device group where each process will get a device. Messages can be communicated between devices of the same device group. Messages cannot be communicated across devices of two different groups.

A device can be freed by the following call

```
LCI_error_t LCI_device_free(LCI_device_t
    *device_ptr);
```

The `device_ptr` is set to NULL.

4.4.2 Endpoints

Communications are done using an *endpoint* as argument. The endpoint is associated with several attributes that determine properties of communications using this endpoint, such as how sends and receives are matched, or how the completion of a communication is signaled.

LCI initialization creates on each process an endpoint `LCI_UR_ENDPOINT`.

A new endpoint is created by the following call:

```
LCI_error_t LCI_endpoint_init(
LCI_endpoint_t *ep_ptr,
LCI_device_t device,
LCI_plist_t plist)
```

`ep_ptr` is a pointer to the created endpoint

`device` specifies the device it is associated with

`plist` is a property list provides the information on the endpoint attributes We describe property lists in Section 4.8 once we discussed the relevant properties.

The endpoint initialization is a collective call involving all the endpoints in the group.

Endpoints at distinct processes are matched in the order of their creation: The k-th endpoint created at a process will communicate with the k-th endpoint created at any other process.

TBD: Is the call barrier-like?

The total number of endpoints at a process cannot exceed `LCI_MAX_ENDPOINTS`. An endpoint is freed with a call to

```
LCI_error_t LCI_endpoint_free(
LCI_endpoint_t *ep_ptr);
```

The call frees the endpoint resources and sets `endpoint` to `NULL`. This is a collective call involving all processes in the same group. It is erroneous to invoke it while there is a pending communication on this endpoint, and it is erroneous to communicate with a freed endpoint.

4.5 Memory Management

LCI communication buffers (see Section 4.6.1) must reside in memory that was registered with the communicating device when they are used from RDMA as source or destination.

This can be done in several ways:

1. User can register a contiguous address range with a device
2. User can request LCI to allocate registered memory. We discuss this method in Section 4.6.
3. Memory can be registered on the fly when a relevant communication call occurs.

The third option provide more flexibility, at the expense of some additional overhead.

A contiguous address range is registered with a call to

```
LCI_error_t LCI_memory_register(
LCI_device_t device,
void *address,
size_t length,
LCI_reg_t *segment);
```

The input arguments are
`endpoint`, the device the memory range is registered with
`address`, the starting address of the registered memory range
`length`, the size in bytes of the registered address range
The call returns in `segment` a registration structure for the memory range
Multiple memory ranges can be registered to the same device, and a memory range may be registered to multiple devices.

> **Implementation Note:** The registration data structure holds the keys that that is needed to use locations within the segment for RDMA communication.

A memory range is deregistered with the call

```
  LCI_error_t LCI_memory_deregister(LCI_segment_t *
       segment )
```

Registered memory must be deregistered before it is freed.

A registration with the special value `LCI_SEGMENT_ALL` is used to indicate in communication calls that the communication buffer should be registered on-the-fly. The buffer will be registered by the communication call so as to enable the communication. It will be transparently deregistered before the memory is freed. Regular registrations are persistent: They persist until the user call `LCI_memory_deregister`. Registrations done for memory marked with `LCI_SEGMENT_ALL` are transient: Such memory can be deregistered transparently from the user. (This does not affect the semantics of LCI, since such deregistered memory will be reregistered on the fly, if needed; but performance may be affected.)

> **Implementation Note:** On-the-fly registration can be done either using a caching mechanism and trapping system calls that unmap physical memory, or by using the caching option in Infiniband.

> ☞ On-the-fly registration adds overhead to communication, especially on-the-fly registration of the target of a put, as it forces an additional rendezvous.

4.5.1 Communicating segment registrations

The use of RDMA communication may require segment registrations to be communicated between processes. To support this, LCI provides functions to serialize and deserialize segment registrations. The function

```
        LCI_error_t LCI_segment_serialize (
          LCI_segment_t      segment ,
          char * string);
```

serializes the `segment` into the string `string`.

```
        LCI_error_t LCI_segment_deserialize (
        char * string,
        LCI_segment_t * segment );
```

reverses the operation.

4.6 Communication buffers

4.6.1 Long buffer

A *long* communication buffer Is used for RDMA communication; it is not copied by LCI, irrespective of its size. A long buffer is specified by a structure with three fields

```
typedef  LCI_lbuffer_t {
    LCI_segment_t segment;
    void* address;
    size_t length; }  LCI_lbuffer_t;
```

`segment` is a registration for a memory range that contains the buffer.
`address` is the buffer's starting length
`length` is the buffer's length, in bytes
A long buffer can be allocated by a call to

```
LCI_error_t LCI_lbuffer_alloc(
    LCI_device_t    device,
    size_t  size,
    LCI_lbuffer_t * lbuffer )
```

The call allocates the required memory and register it with the specified device, returning a long buffer descriptor in `lbuffer`.

> **Implementation Note:** This is a convenience function that calls `malloc`, next `register`.

A long buffer can also be allocated by invoking the function below:

```
 LCI_error_t LCI_lbuffer_memalign   (
    LCI_device_t    device,
    size_t  size,
    size_t  alignment,
    LCI_lbuffer_t *     lbuffer)
```

This same as `LCI_lbuffer_allocate`, except that the buffer is memory alligned as specified by the `aligned` argument.

4.6.2 Medium buffer

Medium buffers have fixed size and are residing in registered memory managed by LCI. The environment parameter `LCI_MEDIUM_SIZE` stores the size of medium buffers. A medium buffer is described by a structure with the following fields:

```
    typedef  LCI_mbuffer_t {
        void* address;
        size_t length;
        }  LCI_mbuffer_t;
```

`address` is the starting address of the buffer.

`length` is the number of significant bytes in the buffer. The data is stored in bytes `address, ..., address+length-1`.

A medium message may involve additional memory-to-memory copying if the source or destination of the communicated data are not in an LCI-allocated medium buffer. not in an LCI medium buffer.

A medium buffer is allocated by a call to

```
LCI_error_t LCI_mbuffer_alloc(
    LCI_device_t    device,
    LCI_mbuffer_t *     mbuffer )
```

The buffer can be used for communication via the specified `device`.

The buffer is freed via a call to

```
        LCI_error_t LCI_mbuffer_free(   LCI_mbuffer_t
             mbuffer )
```

4.6.3 Short data

A short message consists of `LCI_SHORT_SIZE` bytes. Such a message is passed to communication calls by value, rather than by reference. A short message descriptor has the following type.

```
typedef struct LCI_short_t {
    char  val[LCI_SHORT _SIZE];
    } LCI_short_t;
```

4.6.4 IO Vectors

An IO vector is used to send one medium buffer and multiple long buffers in one communication operation. Such message is described by a structure of the following type

```
typedef struct LCI_iovec_t {
    LCI_mbuffer_t piggy_back;
    LCI_lbuffer_t lbuffers;
    int count;
}
```

The second argument is an array of long buffer descriptors, or length `count`. The maximum length of this array is defined by `LCI_IOVEC_SIZE`.

The first argument is a handle to a medium buffer. The maximum size of this buffer depends on the number of long buffers in the I/O vector list. This can be queried with a call to

```
size_t LCI_get_iovec_piggy_back_size(
   int  count);
```

`count` is the number of long buffer in the io vector.

4.7 Completion Objects

Many LCI communication calls are *split*, or *asynchronous*, with the call returning before the communication is locally complete. LCI provides three communication mechansims to indicate the local completion of a communication: *completion queues*, *synchronizers* and *handlers*. A queue enqueues information on completed communications. A synchronizer changes state when a communication is complete; and a handler is invoked when the communication is complete. The type of completion mechanism used is specified by the endpoint argument of the communication call.

In each of these cases, a *request* data structure is used to provide information on the completed operation. The request content can also be used to identify which communication was completed.

Implementation Note: Operations on completion objects may block or unblock invoking threads. The LCI library provides a different implementation for each thread library it interfaces with. This is the main dependence of LCI on the thread package functionality.

4.7.1 Request

A *Request* is used by the various completion mechanisms to return information on a completed communication. A request is a structure of type `struct LCI_request_t`, as defined below.

```
typedef struct {
    LCI_error_t flag;
    uint32_t rank;
    uint16_t_t tag;
    enum LCI_data_type_t type;
    LCI_data_t data;
   void *user_context;
} LCI_request_t
```

`flag` indicates success or failure of the communication associated with this request.

`rank` is the rank of remote participant in a point-to-point communication.

`tag` is the tag transferred with the message. The maximum value of a tag is given by `LCI_MAX_TAG`.

`type` provides information about the type of buffer used. It is of a type defined as

```
typedef enum {
    LCI_IMMEDIATE ,
    LCI_MEDIUM.
    LCI_LONG,
    LCI_IOVEC
} LCI_data_type_t;
```

`data` either contains received short data or a handle to a buffer.

This argument has type `LCI_data_t` defined as

```
typedef union {
    LCI_short_t immediate;
    LCI_mbuffer_t mbuffer;
    LCI_lbuffer_t lbuffer;
    LCI iovec_t iovec;
} LCI_data_t;
```

`user_context` is an additional parameter provided by users that helps differentiate messages when they complete locally.

When a request indicates local completion of a send, a put or get, then `flag`, `rank` and `tag` are the only significant fields. When a request indicates the completion of a receive or the local completion of a put or get initiated by another process, then all fields are significant.

Requests are allocated by the user. LCI fills a user-provided request structure when an entry is deleted from a completion queue, or a synchronizer is set by a completion event; LCI also passes by value requests to handlers. There is no transfer of ownership of request structures between user and LCI. LCI also needs to maintain internally request-like data structures for storing information on completed messages. This information is passed to the user code by copying the information from an internal LCI request data structure to a user request data structure.

Rationale: MPI-defined objects normally are *opaque*: Their structure is implementation-defined and they are created and accessed using MPI-defined methods. This provides implementation flexibility and additional safety. We could use the same approach for requests – by defining multiple request query functions and inlining them. The resulting code will likely be more verbose and somewhat less efficient.

4.7.2 Completion Queue

A *Completion Queue* is (logically) a queue of requests. It has type `LCI_comp_t`.

A completion queue is created by a call to

```
LCI_error_t LCI_queue_create(
    LCI_device_t  device,
    LCI_comp_t *cq);
```

The call specifies the devie the queue is associated with and returns a point to the created queue in `cq`. The initial size of a queue (the number of entries it can store) is set by `LCI_DEFAULT_QUEUE_LENGTH`. The queue may grow, up to `LCI_MAX_QUEUE_LENGTH`. A queue is created at initialization time, with the handle `LCI_UR_CQ`.

The completion queue is freed by a call to

```
LCI_error_t LCI_queue_free(
    LCI_comp_t *cq);
```

The call sets the `cq` argument to `NULL`.

A completion queue supports the following public methods:

```
LCI_error_t LCI_queue_pop(
    LCI_comp_t *cq,
    LCI_request_t *req);
```

The call returns the first entry in the queue. If the queue is empty then it returns a NULL pointer. The entry is deleted from the queue and consumed when the call returns.

`cq` is a handle to the completion queue

`req` is a handle to a request allocated by the user. LCI will copy the information on the completed communication into the user request.

```
LCI_error_t LCI_queue_pop_multiple(
    LCI_comp_t* cq,
    size_t  request_count,
    LCI_request_t request[]),
    size_t  return_count);
```

If the queue contains k entries for completed communications, then the call will return `return_count` = min(`k`, `count`) entries. The call returns the corresponding `return_count` request in the `request[]` array (it returns NULL ias `return_count = 0`) and returns `request_count` in the last argument. first entries in the queue. If the queue is empty then it returns a NULL pointer. The entries are deleted from the queue and consumed when the call returns.

☞ The user can get all the requests in the queue by setting `request_count` to `SIZE_MAX`.

The two `queue_pop` calls are nonblocking: They return immediately. The next two `queue_wait` calls are blocking: If the queue does not contain the required entries, then the calling thread yields and is rescheduled when the request can be satisfied.

```
LCI_error_t LCI_queue_wait (
    LCI_comp_t  cq,
    LCI_request_t *      request );
```

The calling thread blocks until the queue is not empty. The call completes when the thread is awoken; the call returns the first entry in the queue and deletes it from the queue.

If multiple threads block concurrently on the same queue, then the order they are woken up is arbitrary. Each one is returned a different entry from the queue.

```
LCI_error_t LCI_queue_wait_multiple(
     LCI_comp_t queue,
     size_t request_count,
     LCI_request_t requests[]);
```

The calling thread blocks until the completion queue contains `request_count` requests, then the call will return (copy) these requests into the array `requests`.

```
LCI_error_t LCI_queue_len(
     LCI_comp_t queue,
     size_t *len)
```

Returns the number of entries in the queue.

The completion queue has a private push method that is used by LCI: If communication uses a completion queue then upon completion of a communication at the producer side a request is enqueued in the appropriate queue; the request contains the message destination rank, tag and user context; the contents of the `type data` fields are undefined. Upon completion of a communication at the consumer side, a request is enqueued that contains source rank, tag and type; and the message data for an immediate message, or the buffer address and received data length in bytes, otherwise. `flag` is set to 0 if the communication succeeded, or to a nonzero value indicating failure and its cause. A completion queue can be associated with only one device; it can be associated with multiple endpoints on the same device.

4.7.3 Synchronizer

A *synchronizer* is used by user code to poll or wait for communication events to occur and by LCI to signal communication completion.

A synchronizer is created by a call to

```
LCI_error_t LCI_sync_create(
    LCI_device_t device,
    int threshold,
    LCI_comp_t *synchronizer);
```

The `device` argument specifies the device the synchronizer is associated with.

The `threshold` argument specifies how many communication completions trigger the synchronizer. `threshold` should be lesser or equal to `LCI_MAX_SYNC_LENGTH`.

A handle to the synchronizer is returned in `synchronizer`.

A synchronizer is freed by a call to

```
LCI_error_t LCI_sync_free(LCI_comp_t *synchronizer);
```

The call sets the pointer to `NULL`.

A synchronizer with threshold $= k$ enables one task to wait for k communications to complete. The synchronizer "fires" when one task is waiting and k communications have completed. At that point, the waiting task unblocks and is marked as ready. The behavior is undefined if more than k messages complete using this synchronizer while no task is waiting, or if two tasks wait on the same synchronizer before k communication complete. The synchronizer is reset after it fires. The behavior of a synchronizer, with threshold $k = 3$, is illustrated in Figure 1.

The following methods are used to access a synchronizer:

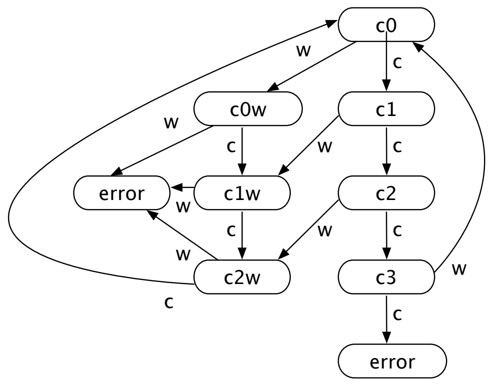

Figure 1: Transition state diagram for a synchronizer with threshold 3. "c" indicates the completion of a communication and "w" indicates a wait call.

```
LCI_error_t LCI_sync_wait(
    LCI_comp_t sync,
    LCI_request_t request[]);
```

`sync` is a synchronizer handle. If this synchronizer was created with `threshold`=k, then the calling task blocks until *k* communications signaled completion using this synchronizer. At this point, the task is marked as ready, and the synchronizer is reset. If *k* communications have already completed then the call returns immediately. The communications can complete before or after the wait is posted; but they must complete after the synchronizer was reset.

The call passes an array of requests in the `request[]` argument and LCI returns in the first `threshold` entries in the array information on the completed communications.

Synchronizers can also be polled:

```
LCI_error_t LCI_sync_test(
    LCI_comp_t sync,
    LCI_request_t request[]);
```

The call returns`LCI_ERR_RETRY` if fewer than `threshold` communications completed with this synchronizer, Otherwise, it returns `LCI_OK`,

Synchonizers are normally triggered by message completions. However, it is also possible to trigger synchronizers by invoking the method below:

```
LCI_error_t LCI_sync_signal(
    LCI_comp_t sync,
    LCI_request_t request);
```

This call will have the same effect as if a communication with the parameters specified by the `request` arguments signaled completion using the synchronizer; the values provided in the `request` argument will be subsequently returned by the corresponding `LCI_sync_wait` or `LCI_sync_test` call.

☞ LCI does not validate the content of the `request` argument passed to `LCI_sync_signal` and return it as is to the subsequent `LCI_sync_wait` or `LCI_sync_test`; the content can be arbitrary.

4.7.4 Handlers

A handler is a function that may be invoked when a completion event occurs. The environment where the function executes is not specified, and handler invocations might be concurrent. So a handler should be thread-safe. It should not execute any LCI call.

A handler has the following type:

```
typedef void (*LCI_handler_t)(
    LCI_request_t request);
```

A handler is associated with a device by the following call:

```
LCI_error_t LCI_handler_create(
    LCI_device_t device,
    LCI_handler_t handler,
    LCI_comp_t *completion);
```

The argument `device` specifies the device the handler is attached to.
`handler` is a pointer to the handler function.
The call returns in `completion` a handle to the handler.

☞ A variable of type `LCI_comp_t` should be cast to to type `uintptr_t` before it is communicated across processes.

4.8 Property Lists

Property lists are mechanisms for setting the properties of an endpoint when it is created. They are objects of type `LCI_plist_t`. These arew created by a call to

```
LCI_error_t LCI_plist_create(LCI_plist_t *plist_ptr);
```

They are freed by a call to

```
LCI_error_t LCI_plist_free(LCI_plist_t *plist_ptr);
```

The call sets the `plist_ptr` argument to `NULL`.

```
LCI_error_t LCI_plist_get(
    LCI_endpoint_t endpoint,
     LCI_plist_t *plist);
```

Gets the property list attached to an endpoint.

```
LCI_error_t LCI_plist_decode(
    LCI_plist_t plist
     char *string);
```

Returns a string that decodes the property list `plist` in a human-readable from into `string`.

The following properties can be attached to an endpoint

4.8.1 Matching Mechanism

For two-sided communication, the user may specify whether matching on the (local) receive side is by source rank and tag, or by tag only. This is done with the call below

```
LCI_error_t LCI_plist_set_match(
    LCI_plist_t *plist,
    LCI_match_t match_type);
```

The types are defined by the `LCI_match_t` enumeration.

```
typedef enum {
    LCI_MATCH_RANKTAG = 0,
    LCI_MATCH_TAG
} LCI_match_t;
```

If no matching type is specified, then the default is matching by sender's rank and tag. An endpoint can be associated with only one matching mechanism.

☞ Matching by tag only can be used to support the equivalent of `MPI_ANY_SOURCE` wildcard in MPI – except that wildcard receives and regular receives do not mix. Matching by rank only is not necessary as it can be achieved by using a fixed tag value.

4.8.2 Completion Mechanism

```
LCI_error_t LCI_plist_set_comp_type(
    LCI_plist_t plist,
    LCI_port_t port,
    LCI_comp_type_t type);
```

Sets the completion mechanism for the specified port in the property list. `LCI_port_t` is defined as

```
typedef enum {
    LCI_PORT_COMMAND = 0,
    LCI_PORT_MESSAGE = 1,
} LCI_port_t;
```

If `port` equals `LCI_PORT_COMMAND` (0) then the completion method is specified for the command port, i.e., for signaling completion of locally issued communication calls. If `port` equals `LCI_PORT_MESSAGE` (1) then the completion method is used for the completion of a communication action triggered by the arrival of a message.

`LCI_comp_type_t` is defined as

```
typedef enum {
    LCI_COMPLETION_NONE = 0,
    LCI_COMPLETION_QUEUE,
    LCI_COMPLETION_HANDLER,
    LCI_COMPLETION_SYNC
} LCI_comp_type_t;
```

☞ `LCI_COMPLETION_NONE` is not currently useful as LCI does not provide a method to check communication completion, other than via one of the three completion mechanisms. It is provided for future extensions.

4.8.3 Default Completion Object

For one-sided communication, the user must specify a default completion object (a synchronizer, completion queue, or handler). All the completion events on the target side will be reported to the default completion object of the corresponding endpoint. The default completion object is set by

```
LCI_error_t LCI_plist_set_default_comp(
    LCI_plist_t plist,
    LCI_comp_t comp);
```

The default completion object specified by this function must be consistent with the completion type of the message port specified by `LCI_plist_set_comp_type`.

4.9 Two-Sided Communication

4.9.1 Long

Send long sends a long buffer

```
LCI_error_t LCI_sendl(
    LCI_endpoint_t ep,
    LCI_lbuffer_t buffer,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The `ep` argument specifies the local endpoint for this call. The message can received at the matching destination endpoint.

The `buffer` argument specifies the local buffer, i.e., the containing segment, the start address and the length. The segment must be registered with the endpoint's device, unless the segment registrations is `LCI_SEGMENT_ALL`.

The `rank` argument specifies the rank of the destination.

The `completion` argument is a handle for a local queue, synchronizer or handler; the type of completion mechanism has to match the completion type specified for the command port of the local endpoint.

Local completion is signaled when the source buffer can be reused by the application.

☞ send operations do not necessarily complete in the order they were initiated. The user can match send calls to completion events by checking the values of `rank` and `tag` in the returned request. It is legal to reuse the same values of `destination, tag` for successive sends, before testing or waiting for their completion; but, then, the user cannot distinguish which send has completed, unless they carry a distinct `user_context` field.

A send long is matched by the following *receive long*.

```
LCI_error_t LCI_recvl(
    LCI_endpoint_t endpoint,
    LCI_lbuffer_t buffer,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The message is received in the buffer represented by the argument `buffer`. The arguments `rank` and `tag` are used to match send to receive. A receive matches a send if the matching type of the message post of the receiving endpoint is set with `LCI_MATCH_RANKTAG`, sender rank matches receiver `rank` argument, and sender `tag` argument matches receiver `tag` argument; if the matching type of the message port of the receiving endpoint is set with `LCI_MATCH_TAG` then only tags need to match, and the `rank` argument is ignored.

Messages are not necessarily matched in the order they are sent. The same matching logic is used for all send-receive calls.

If the argument `buffer` passed to `recvl` has the address `NULL`, the receive buffer will be allocated by LCI in the specified segment. The address and length of the receive buffer will be returned in a request via the completion mechanism (which cannot be `NULL`). The user needs to free the LCI-allocated memory using the `LCI_lbuffer_free` call.

Long message 2-sided communication uses a rendezvous protocol: The sender issues a "request to send"; when a matching receive is posted, the receiver (optionally) issues an "ready to receive" reply or issues an rDMA get. An rDMA transfer from source buffer to destination buffer (either put or get) is then performed. The request to send includes the message length that is needed for buffer allocation. If the `buffer`'s address is NULL and memory cannot be allocated, LCI will treat it as a fatal error and abort.

Rationale: As the name indicates, the use of a long send is recommended for long messages, as the rendezvous protocol adds to the fixed overhead, but avoids a copy of the data, and reduces memory consumption. Implementations will specify a suggested threshold above which the use of a long send is desirable. But the proper choice may depend on various factors, such as the amount of memory pressure, the relative pressure on sender and receiver, etc. Therefore, LCI let the user choose the protocol.

4.9.2 Medium

Medium buffer send calls have two forms.

The first one, *send medium*, is

```
LCI_error_t LCI_sendm(
    LCI_endpoint_t ep,
    LCI_mbuffer_t buffer,
    int rank,
    LCI_tag_t_t tag);
```

The semantic is same as for `LCI_sendl`, with one difference: The call is elemental and returns only after the input buffer can be reused. Hence, no completion argument is passed. The communication buffer can be anywhere in memory, and its length should be shorter or equal to the length of a medium message.

The second call, *send medium nocopy* is as follows:

```
LCI_error_t LCI_sendmn(
    LCI_endpoint_t ep,
    LCI_mbuffer_t buffer,
    int rank,
    LCI_tag_t tag);
```

This call behaves as `LCI_sendm`, except that the buffer is a medium buffer provided by LCI. This call also is elemental (returns immediately) but the buffer is returned to LCI and cannot be reused by the user; the user has to acquire a new buffer with a call to `LCI_mbuffer_alloc`.

Rationale: A call to LCI_sendm is typically implemented by copying the user buffer into an internal LCI buffer, from where it is sent. A call to LCI_sendmn may avoid the extra copy. LCI_sendmn does not return the buffer to the caller, because the communication may be delayed, but the call needs to return immediately. A call to sendm can be implemented by a call to allocate a medium buffer, followed by copying the user data to the allocated buffer, followed by a call to sendmn

There are two *medium receive* calls.

The first is *receive medium*:

```
LCI_error_t LCI_recvm(
    LCI_endpoint_t ep,
    LCI_mbuffer_t *buffer,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The received message is copied into the receive buffer specified by the call. The buffer can be anywhere in the address space.

The second is *receive medium nocopy*:

```
LCI_error_t LCI_recvmn(
    LCI_endpoint_t ep,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The incoming message is buffered into a fixed-size medium LCI buffer; the address of this buffer and the number of significant bytes in the buffer are returned via the completion mechanism, which cannot be null. The application needs to return eventually the buffer to LCI with a call to LCI_medium_free.

LCI_sendm, LCI_sendmn can each be matched either by LCI_recvm or LCI_recvmn.

LCI uses an eager send protocol for medium messages: The message is sent as soon as the send is posted, with no handshake with the destination. This avoids the need for an handshake, but may involve extra copying of the data and source and/or destination. While completion at the source does not imply that the communication has completed at the destination, it implies that the communication will eventually complete (assuming that a matching receive is eventually posted).

☞ If a medium message is received in a user specified buffer, then the number of bytes updated equals the number of bytes sent. If a message of n bytes is received in a LCI-allocated medium buffer, then the first n bytes will contain the received message, but the values of the other bytes in the message are undefined.

Implementation Note: The implementation must use a reliable protocol and ensure that buffer space is available on the receiver to buffer incoming messages.

TBD: We can merge `recvm` and `recvmn` by passing a null buffer address when LCI is to use an LCI-allocated buffer.

4.9.3 Short

A *send short* has the following syntax

```
LCI_error_t LCI_sends(
    LCI_endpoint_t ep,
    LCI_short_t data;
    int rank,
    LCI_tag_t tag);
```

The call is elemental.

A `send short` is matched by a *receive short* that has the following syntax.

```
LCI_error_t LCI_recvs(
    LCI_endpoint_t ep,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The sent value is returned via the completion mechanism.

4.9.4 IO Vector

```
    LCI_error_t LCI_sendv(
    LCI_endpoint_t ep,
    LCI_iovec_t iovector,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

This call has the same behavior as a `LCI_sendl` call, except that the source buffer is an IO vector.

An IO vector send call is matched by an IO vector receive call with the following syntax:

```
    LCI_error_t LCI_recvv(
    LCI_endpoint_t ep,
    LCI_iovec_t iovector,
    int rank,
    LCI_tag_t tag,
    LCI_comp_t completion,
    void *user_context);
```

The receive IO vector must be *compatible* with the send IO vector: Both should contain the same number of long buffers, with each receive long buffer being at least as long as the corresponding send long buffer. If the receive iovector argument is NULL, then the receiver will allocate the long buffers required to receive the matched message.

4.10 One-Sided Communication

4.10.1 Long

A *put long* call has the syntax

```
LCI_error_t LCI_putl(
    LCI_endpoint_t ep,
    LCI_lbuffer_t local_buffer,
    LCI_comp_t local_completion,
    int rank,
    LCI_tag_t tag;
    LCI_lbuffer_t remote_buffer,
    uintptr_t remote_completion,
    void *user_context);
```

`local_buffer` specifies the containing segment, starting address and length of the source buffer.

`tag` is used as an identifier for the communication.

`local_completion` is the handle to a completion object in local memory, or `NULL`.

`remote_buffer` specifies the containing segment, starting address and length of the destination buffer. The segment must be registered with the remote communicating device; i.e., `LCI_SEGMENT_ALL` cannot be used.

`remote_completion` is a handle to a completion object in remote memory or `NULL`. If it is equal to `LCI_DEFAULT_COMP_REMOTE` then the default completion on the remote endpoint is used.

The put operation copies the local buffer content to to the remote buffer – typically, using RDMA. The transfer can start as soon as the call is made. The local completion event will indicate that the local buffer can be reused. The remote completion event will indicate that the remote buffer was updated and can be accessed.

☞ We use `uintptr_t` as the type for a remote pointer. This is to flag that this is not a reference that can be dereferenced locally.

☞ The calling process has to receive from the destination a segment descriptor and an address, in order to to set locally the fields of the `remote_buffer` data structure. The segment descriptor is passed the serialize/deserialize methdos for segement descriptors; the address should be passed as a value of type `uintptr_t`.

A *put long allocate* call has the syntax

```
LCI_error_t LCI_putla(
    LCI_endpoint_t ep,
    LCI_lbuffer_t local_buffer,
    LCI_comp_t local_completion,
    int rank,
    LCI_tag_t tag,
    uintptr_t remote_completion,
     /* Currently, LCI  only supports
        the  value LCI_DEFAULT_COMP_REMOTE
        for this argument  */
    void *user_context);
```

It is similar to `putl`, except that the producer does not pass to LCI a destination buffer. The remote buffer will be allocated by LCI and the buffer address will be returned via the completion mechanism. Successful completion at the source indicates that the destination buffer is allocated and that the local buffer can be reused. The destination process should eventually return the buffer to LCI by using the function `LCI_lbuffer_free`.

☞ LCI currently supports only `remote_completion = LCI_DEFAULT_COMP_REMOTE` for one-sided communication

The implementation of this put function uses a rendezvous protocol: A "request-to-send" is sent to the destination; `LCI_lbuffer_alloc` is invoked at the target side and is passed the `length` argument. An "OK-to-put" (or "allocation failed") message is returned to the source. Accordingly, the source either proceeds with the RDMA transfer, or returns a failure indication through the local completion mechanism.

A *get long* has the following syntax

```
LCI_error_t LCI_getl(
    LCI_endpoint_t ep,
    LCI_lbuffer_t *local_buffer,
    size_t  length,
    LCI_comp_t local_completion,
    int rank,
    LCI_tag_t tag,
    LCI_lbuffer_t remote_buffer,
    uintptr_t remote_completion,
    void *user_context);
```

Data is copied from the remote buffer to the local buffer. Remote completion occurs when the remote buffer can be reused and local completion occurs when the local buffer has been filled and can be accessed.

The remote buffer must be in memory registered with the relevant device.

4.10.2 Medium

A *put medium* has one of the four following forms, for all the possible combinations of the type of source buffer (user-provided or LCI-allocated) and destination buffer (user provided or LCI-allocated). The call is elemental and returns when the local buffer can be reused.

Put medium:

```
LCI_error_t LCI_putm(
    LCI_endpoint_t ep,
    LCI_mbuffer_t buffer,
    int rank,
    LCI_tag_t tag,
    LCI_mbuffer_t remote_buffer,
    uintptr_t remote_completion);
```

The call specifies both the local buffer and the remote buffer. They can be anywhere in user memory.

Put medium allocate

```
LCI_error_t LCI_putma(
    LCI_endpoint_t endpoint,
    LCI_mbuffer_t local_buffer,
    int rank,
    LCI_tag_t tag,
    uintptr_t remote_completion
    /* Currently, LCI  only supports
    the  value LCI_DEFAULT_COMP_REMOTE
    for this argument  */
    );
```

The call specifies the local buffer, that can be anywhere in local memory. The remote medium buffer is allocated by LCI and will need to be returned by the user. The allocated buffer has fixed size.

Put medium nocopy

```
LCI_error_t LCI_putmn(
    LCI_endpoint_t ep,
    LCI_mbuffer_t buffer,
    int rank,
    LCI_tag_t tag,
    LCI_mbuffer_t remote_buffer,
    uintptr_t remote_completion);
```

Same as `putm`, except that the local buffer is an LCI-allocated buffer.

Put medium nocopy allocate

```
LCI_error_t LCI_putmna(
    LCI_endpoint_t ep,
    LCI_mbuffer_t buffer,
    int rank,
    LCI_tag_t tag,
    uintptr_t remote_completion
    /* Currently, LCI  only supports
    the  value LCI_DEFAULT_COMP_REMOTE
    for this argument  */
    );
```

Both local buffer and remote buffer are LCI-allocated.

TBD: One could leave LCI figure out whether the local buffer is an LCI-allocated one or user allocated one. Could use a NULL remote buffer to indicate that it needs to be allocated.

4.10.3 Short

```
LCI_error_t LCI_puts(
    LCI_endpoint_t ep,
    LCI_short_t data,
    int rank,
    LCI_tag_t tag,
    uintptr_t remote_completion);
```

This call is elemental.

4.10.4 IO Vector

An IO vector put has the following form

```
    LCI_error_t LCI_putvs(
    LCI_endpoint_t ep,
    LCI_iovec_t local_iovec,
    int rank,
    LCI_tag_t tag,
    LCI_iovec_t remote_iovec,
    uintptr_t remote_completion,
    void* user_context);
```

The `local_iovvec` must be compatible with the `remote_iovec`. The piggy-back buffer can be immediately reused. the local long buffers can be reused only after local completion.

An IO vector put-allocate has the following form:

```
LCI_error_t LCI_putva(
LCI_endpoint_t ep,
LCI_iovec_t iovec,
LCI_comp_t completion,
int rank,
LCI_tag_t tag,
uintptr_t remote_completion,
/* Currently, LCI  only supports
the  value LCI_DEFAULT_COMP_REMOTE
for this argument  */
void* user_context);
```

An IO vector get has the form

```
LCI_error_t LCI_getv(
    LCI_iovec_t local_iovec,
    int rank,
    LCI_tag_t tag,
    LCI_iovec_t remote_iovec,
    uintptr_t remote_completion
    uintptr_t remote_completion,
    /* Currently, LCI  only supports
    the  value LCI_DEFAULT_COMP_REMOTE
    for this argument  */
    void* user_context);
```

4.11 Bulk Completion

☞ Bulk completion is not yet implemented.

```
LCI_error_t LCI_fence_wait(
    LCI_endpoint_t endpoint)
```

The calling thread blocks until all communications initiated at the local process on this endpoint have completed locally. The fence call does not complete remote calls that communicate with the local process.

```
LCI_error_t LCI_gfence_wait(
    LCI_endpoint_t endpoint)
```

The calling thread blocks until all communications initiated by endpoints in the group of the endpoint have completed, both locally and remotely.

4.12 Progress

```
LCI_error_t LCI_progress(LCI_device_t device);
```

A call to this function effects polling on the specified device. The function is nopt thread-safe.

☞ Only one thread can effect progress on a device at a time.

References

[1] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[3] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[4] Hoang-Vu Dang, Marc Snir, and William Gropp. Eliminating contention bottlenecks in multithreaded mpi. *Parallel Computing*, 69:1–23, 2017.

[5] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768. ACM, 2018.

[6] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[7] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D Russell, Howard Pritchard, and Jeffrey M Squyres. A brief introduction to the Openfabrics interfaces-a new network API for maximizing high performance application efficiency. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 34–39. IEEE, 2015.

[8] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[9] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *OOPSLA*, pages 91–108. ACM, 1993.

[10] Gregory F Pfister. An introduction to the Infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[11] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran 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*, pages 40–43. IEEE, 2015.

[12] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.