Design and Analysis of the Network Software Stack of an Asynchronous Many-task System – The LCI parcelport of HPX

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

Hartmut Kaiser
hkaiser@cct.lsu.edu
Louisiana State University
Baton Rouge, LA, USA

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

ABSTRACT

The HPX asynchronous many-task runtime system has been using TCP and MPI as its communication backends (parcelports). We developed a new HPX parcelport using a new communication library, the Lightweight Communication Interface (LCI) that was designed to better match the needs of systems such as HPX. We evaluate its performance with various microbenchmarks and a real-world astrophysics application, Octo-Tiger. Compared to the best configuration of the MPI parcelport, microbenchmarks show that the new LCI parcelport improves the message rate by up to 30x and decreases latencies by up to 5x. It also reduces the total execution time of Octo-Tiger by up to 1.175x compared to the best configuration of the MPI parcelport and up to 13.6x compared to the same configuration of the MPI parcelport. We discuss the performance impacts of different design choices.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; • **Computing methodologies** → **Parallel programming languages**.

KEYWORDS

asynchronous many-task systems, communication libraries, multithreaded message passing

ACM Reference Format:

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 *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624598

1 INTRODUCTION

The Message Passing Interface (MPI) has been the dominant communication library of parallel computing for more than 20 years. Coming with it is the most common way of writing programs on parallel architectures, the Bulk-Synchronous Programming (BSP) model, where all processes progress in lockstep. The BSP model faces challenges when dealing with irregular problems such as graph algorithms and sparse numerical solvers and with heterogeneous architectures.

Asynchronous many-task runtime systems (AMTs), such as PaRSEC [7], Legion [4], StarPU [1], and HPX [15, 16], offer a potential solution to these challenges. In these systems, parallel computations are expressed as a set of fine-grained tasks and dependencies between those tasks. The dependency management, data movement, and task scheduling are performed by the runtime. In this way, one hopes to improve load balancing and achieve better computation-communication overlap. However, the communications of AMTs are typically multithreaded, irregular, and mix small and large messages. This is not the traditional use cases of the communication libraries such as MPI [23]. As a result, the communication software stack will often become the performance bottleneck as task systems scale out.

The HPX asynchronous many-task runtime system [15, 16] extends C++ to enable users to describe arbitrary task dependency graphs for tasks mapped across multiple computation nodes. Prior to this project, it had two communication backends (*parcelports*): TCP and MPI. The Lightweight Communication Interface (LCI) is a communication library and research tool under active development [11, 28]. Its primary goal is to provide efficient support for multithreaded and irregular communications. In this project, we integrate LCI into the HPX communication stacks and create an LCI parcelport for HPX. We evaluate its performance using various microbenchmarks and an application-level benchmark, Octo-Tiger [21], an astrophysics application simulating star merging based on the fast multipole method and adaptive octrees. We also investigate the performance impact of various design decisions that differentiate the LCI parcelport from the MPI parcelport.

The rest of the paper is organized as follows. Section 2 gives a brief overview of LCI and HPX's network stack. Section 3 describes the design and implementation of the existing MPI parcelport of HPX and the new LCI parcelport along with its multiple research variants. Sections 4 and 5 present the experiment results and detailed analysis of the microbenchmarks and the Octo-Tiger benchmark. Section 6 describes the related works. Section 7 concludes the paper by discussing our main lessons and future works.

2 SOFTWARE STACK OVERVIEW

2.1 LCI

The Lightweight Communication Interface (LCI) is designed to be an efficient communication library for multithreaded, irregular communications. It is a research tool to explore design choices for such libraries. It has been used to improve the performance of graph

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624598

analytic applications [11]. We currently focus on improving the performance of AMTs, as they can potentially make a big impact on parallel programming and their communications are naturally multithreaded and irregular. We hope to use LCI to answer research questions such as (a) what is the most efficient way to remove or minimize the thread contentions inside and outside the communication libraries in multithreaded irregular applications; and (b) what is the right level of a communication API for multithreaded irregular applications so that they can utilize most of the performance provided by modern hardware without losing too much programmability.

Currently, LCI is implemented as a mix of C and C++ libraries and is based on two communication backends, libibverbs [20] and libfabrics [24]. LCI has the following major features:

- **Multithreaded performance as the first priority:** We carefully design the internal data structures to minimize interference between threads. We use atomic operations and fine-grained try locks extensively instead of coarse-grained blocking locks. This results in a much better and more stable multithreaded performance.
- **Versatile communication interface:** LCI provides users with various communication primitives and message completion mechanisms. For communication primitives, it offers two-sided send/receive and one-sided put/get. Remote buffers can be specified or allocated dynamically. For completion mechanisms, it offers synchronizers (similar to MPI requests but with the option of multiple producers), completion queues, and function handlers. LCI allows users to combine any communication primitives with almost any completion mechanism. This includes signaling the completion of one-sided operations at the remote node. In this way, it intends to enable users to select the communication primitives and completion mechanisms that best fit the applications.
- **Explicit control of communication behaviors and resources:** Users have access to the internal registered communication buffers and memory registration functions and can control the semantics of send/receive tag matching. They explicitly choose whether to use the eager or rendezvous protocols. All communication primitives are non-blocking and users can decide when to retry in case of temporarily unavailable resources. LCI also gives users an explicit function to make progress on the communication engine. Users can tailor the LCI configuration to reduce software overheads, or just use default settings if LCI is not a performance bottleneck.

2.2 HPX Network Stack

HPX [15, 16] provides an RPC-like communication interface. The HPX application can register functions as HPX *actions*. After that, each *locality* (equivalent to MPI rank) can invoke actions on any locality. The collection of arguments to invoke an action, provided by the source locality, along with some metadata of the action invoked, is packed into an HPX-internal data structure called *parcel*. The HPX runtime aggregates parcels sharing the same destination locality, serializes them, transfers them through the network to the target locality, deserializes them, and invokes the specified actions. Optionally, it also aggregates, serializes, transfers, and deserializes the return values as parcels back to the source locality.

The parcel packing, aggregating, serialization, and deserialization are handled in the layer above the parcelport layer. The responsibility of the parcelport layer is to transfer the serialized parcels to the target locality. To simplify the description, we will call the serialized parcels passed to the parcelport layer an *HPX message*.

An HPX message passed to the parcelport layer consists of the following components:

- A non-zero-copy chunk containing all the small arguments of the serialized parcels and some metadata about the parcels.
- Optionally, multiple zero-copy chunks, each containing a large argument of the serialized parcels.
- A transmission chunk containing the index and length of the arguments. It is only needed when there is at least one zero-copy chunk. Otherwise, the remote locality can deserialize the parcels sorely based on the non-zero-copy chunk.

Whether an argument is small or large is determined by an HPX internal parameter: the zero-copy serialization threshold.

3 DESIGN

3.1 The MPI parcelport

Prior to this project, the HPX parcelport with the best performance was the MPI parcelport. We describe below how the MPI parcelport transfers an HPX message.

Connections: Transferring an HPX message involves transferring one or multiple MPI messages. At a given time, there might be multiple HPX messages being transferred concurrently. To manage the chain of MPI messages associated with one HPX message, the MPI parcelport creates a special object, *connection*, for each HPX message. The source locality will create a sender connection when the upper layer passes down an HPX message. The target locality will create a receiver connection when it receives a header MPI message, as described below. All the following sends and receives will use that connection object.

MPI messages: One HPX message is communicated with the following sequence of MPI messages: One protocol message generated by the parcelport, namely the *header message*, and three types of messages passed by the upper layer, namely one *non-zero-copy chunk message*, one (optional) *transmission chunk message*, and zero or more *zero-copy chunk messages*. The header message contains metadata about the HPX message such as the MPI tag it should use for the follow-up sends and receives, the size of the non-zero-copy chunk, and the existence and size of the transmission chunk. The header message is sent to the target locality with MPI tag 0. On the target locality, the MPI parcelport always has one receive posted with the maximum header size and tag 0. It will repeatedly check whether this receive operation has been completed as one of the background activities, and if it has been completed, decode the header and create a receiver connection for the follow-up receives. If there is at least one zero-copy message, the MPI parcelport will transfer the transmission chunk as a message. If the transmission message and the non-zero-copy chunk message are small enough, they will piggyback on the header message. The maximum size of the header message is set to be the zero-copy serialization threshold.

Tag management: Each sender/receiver connection pair needs a distinct tag for its MPI messages. The MPI parcelport uses an

atomic counter to determine the next tag to use. The tag will wrap around after the MPI tag's upper bound. This assumes the previous connection pair with the same tag value will always be completed before that value is reused. Currently, this assumption seems to be satisfied in all tested HPX programs, but a more robust mechanism is needed. This is left to future work.

Concurrency and synchronization: The maximum number of pending connections is set by an internal HPX parameter which is 8192 by default. For each sender or receiver connection, there will be at most one send and one receive posted at any given time. A new `MPI_Isend` and `MPI_Irecv` will only be posted after the previous one is complete. While the current operation posted by a connection is not completed, the MPI parcelport will put the pending connection into a *connection list*. The pending connection list is protected by an HPX spinlock and will be checked periodically by the worker threads as one of the background activities.

Threads and background work: MPI is initialized in the *MPI_THREAD_MULTIPLE* mode. The communication is not funneled to a dedicated thread. Instead, all the worker threads can initiate a sender connection and send MPI messages. They all invoke the parcelport's background work function when idle. The background work function will (a) check the header message receive for new parcels (b) check the pending connection list in a round-robin manner to see whether posted asynchronous operations made by connections have been completed and make progress on the connections with completed operations.

The original version: the MPI parcelport we present here is actually an improved version. During this project, we applied what we learned when developing the LCI parcelport and improved the original MPI parcelport. There are two differences compared to the original version.

- In the original version, the header message buffer is always statically allocated on the stack and fixed at 512 bytes. In addition, the header message can only piggyback the non-zero-copy chunk message. The current version allocates it dynamically and can piggyback the transmission message.
- In the original version, tags are managed by a tag provider. When a receiver connection completes, it will send a "tag release" message back to the sender carrying the tag and the sender will push the released tag into the tag provider. The tag provider is implemented as a lock-protected vector containing all the free tags. When the vector is empty, it will atomically create new tags by incrementing a shared atomic counter. The current version removes the "tag release" message and the free tag list. It just uses a shared atomic counter.

The two optimizations improve the application (Octo-Tiger) performance by about 20% and the first optimization makes the biggest difference.

3.2 The LCI parcelport

We developed a new LCI parcelport for HPX. There are several differences between the MPI and LCI parcelport, such as the communication primitives, the completion mechanisms, and how to make progress on the HPX network background tasks. These differences are rooted in the different functionalities provided by MPI and LCI.

3.2.1 The Baseline Implementation. We first present the baseline implementation of the LCI parcelport, which is the default setting in HPX and yields the best performance in general.

Connections: Same as the MPI parcelport, we create a sender and receiver connection to handle the chain of sends and receives for each HPX message.

The header message: Same as the MPI parcelport, we create a metadata header message for each HPX message. However, there are several differences: First, we directly assemble the header message in an LCI-allocated buffer so that, for eager messages, we save one memory copy. Second, we transfer the header message using the LCI *dynamic put* communication primitive: The target buffer is allocated by the LCI runtime upon message arrival and an entry is pushed to a pre-configured LCI completion queue.

Follow-up messages: All the follow-up messages follow the same logic as the MPI parcelport. We use medium sends and receives for messages of size below LCI's eager message threshold and long (rendezvous) sends and receives otherwise.

Tag management: Similar to the MPI parcelport, we use an atomic counter to get the next available tag. However, because LCI does not guarantee in-order delivery, we use a distinct tag for each follow-up message, rather than one tag for all messages of the same connection. The safety of this approach follows from the same assumptions and has the same limitations as the MPI parcelport.

Concurrency and Synchronization: Same as the MPI parcelport, the LCI parcelport only posts a new send/receive after the previous one completes. However, the LCI parcelport uses completion queues as the completion mechanism so it does not need a connection list to hold all the pending connections and check them repeatedly in a round-robin manner.

Threads and background work: Same as the MPI parcelport, all worker threads can initiate a sender connection and send LCI messages and call the HPX background work function when idle. However, in the LCI parcelport, the background work function only polls the LCI completion queues for new parcels and pending connections with completed operations. The LCI parcelport creates one dedicated progress thread to make progress on the LCI runtime. It is allocated and managed by the HPX scheduling system and pinned by HPX at core 0.

3.2.2 Variants. Besides the baseline implementation, we also implemented different variants of the LCI parcelport to understand how different features of the parcelport impact performance.

Protocol: The LCI parcelport can be configured to use one of the following communication protocols.

- *putsendrecv*: use the one-sided dynamic *put* for the header message and the two-sided *send* and *receive* to transfer the remaining messages, as described in the baseline implementation.
- *sendrecv*: only use the *send* and *receive* communication primitives. We use *send* to send the header message and receive a header message by always having one *receive* posted and checking for its completion, the same as the MPI parcelport.

Progress Type: The LCI parcelport can be configured to use one of the following ways to call the `LCI_progress` function.

- *rp*: use the HPX resource partitioner to create and pin the progress thread onto a dedicated core, as described in the baseline implementation.
- *worker*: do not create a progress thread. Instead, use HPX worker threads to call the LCI progress function periodically. The progress function is modified to be thread-safe.

Completion Mechanism: The LCI parcelport can be configured to use one of the following completion mechanisms.

- *cq*: Use completion queues as the completion mechanism, as described in the baseline implementation.
- *sync*: Use synchronizers as the completion mechanism. We use a pending synchronizer list and round-robin checking similar to the MPI parcelport. However, we do not change the completion mechanism of the one-sided dynamic *put* on the receive side, as the current LCI version only supports a pre-configured completion queue as the remote completion mechanism for puts.

Send Immediate Optimization: By default, the HPX upper layer interacts with two internal data structures when sending a parcel: the connection cache and the parcel queue. We describe each below:

Connection cache: The HPX upper layer maintains a cache of connections. When it needs to send an HPX message to a specific locality, it will first ask the connection cache for a connection to that locality. If there is no such connection available in the cache, it will create a new one unless the existing connection number reaches the configured max value (8192 by default). When a connection completes its work, it will be returned to the cache. This reduces the memory allocation/deallocation frequency.

Parcel queue: The HPX upper layer maintains a parcel queue for each target locality. To send a parcel, the HPX upper layer will first enqueue the parcel into a parcel queue of its locality. After that, it will dequeue all parcels from that parcel queue, serialize them into an HPX message, and pass it to the parcelport layer. This adds opportunities for message aggregation when multiple threads push parcels to the parcel queue simultaneously or (in rare cases) the connection cache runs out of free connection objects.

These two data structures improve aggregation and memory usage. However, accesses to each of those are protected by HPX spin locks so their use also increases lock contention. The MPI and LCI parcelports can use two distinct configurations:

- *default:* Both data structures are used.
- *immediate:* The HPX upper layer serializes directly the parcel into an HPX message and passes it to the parcelport layer, bypassing the connection cache and the parcel queue.

4 MICROBENCHMARKS

We designed a series of microbenchmarks to study systemically the performance of the MPI parcelport and the LCI parcelport and the impact of different design decisions for the LCI parcelport.

The messages of task systems tend to be of small size and thus typically do not saturate the network bandwidth. As a result, we design our microbenchmarks to stress the underlying network stack with relatively small messages (up to 64 KiB) and focus on latency and message rate, instead of the latency and bandwidth that is typically studied in traditional communication microbenchmarks.

All the microbenchmarks use two HPX processes, one sender and one receiver, running on two separate compute nodes, and each uses all the cores on its node. We rely on the task scheduler in HPX to run the communication tasks so messages can be sent and received on any thread.

Table 1 explains the naming scheme used for the configurations tested. In most cases, all the LCI parcelport variants without the send immediate optimization exhibit similar results so we only show one such configuration, *lci_psr_cq_pin*, to reduce clutter. When using both the *putsendrecv* protocol and the *sync* completion type, the completion mechanism for the header messages on the receive side is still a completion queue, as the current implementation of the LCI *put* can only use a pre-configured completion queue as the remote completion object.

All experiments in this section are performed on the SDSC Expanse system. Its configuration is shown in Table 2. Unless otherwise noted, all experiments are performed at least five times and the figure shows the average and standard deviation. We keep the HPX zero-copy serialization threshold at its default value: 8192 bytes.

Table 1: Abbreviations for configurations.

Abbreviation	Configuration
mpi	Use the MPI parcelport
lci	Use the LCI parcelport
sr	Use the sendrecv protocol
psr	Use the putsendrecv protocol
sy	Use synchronizer as the completion type
cq	Use completion queue as the completion type
pin	Use a pinned dedicated progress thread
mt	Use all worker threads to make progress
i	Enable the send immediate optimization

Table 2: SDSC Expanse System Configuration.

CPU	AMD EPYC 7742 64-Core Processor (2 sockets, 128 cores per node)
Memory	256 GB, DDR4
Storage	1TB Local Intel NVMe SSD
NIC	Mellanox ConnectX-6
Interconnect	HDR InfiniBand (2x50Gbps)
Max Allowed Nodes/Job	32 Nodes
OS	Rocky Linux 8.7
Compiler	GCC 10.2.0
Software	OpenMPI 4.1.5, UCX 1.14.0

4.1 Message Rate

For the message rate microbenchmark, a sender attempts to create tasks at a fixed rate. Each task will inject a batch of fixed-size messages. The receiver waits for all messages to be received and then signals back to the sender with one short message. The sender

measures the *achieved injection time* – the time it took to generate all tasks, and the *actual communication time* – the time it took to have all messages received. The two times diverge when messages cannot be sent fast enough, resulting in blocked tasks or queued messages. Rather than timings, we plot the *achieved injection rate* and the *message rate* that are obtained by dividing the number of transferred messages by the achieved injection time and the communication time.

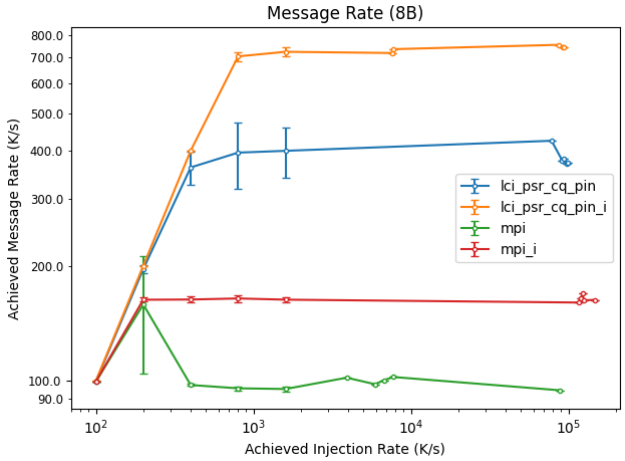

Figure 1: Achieved message rate of 8B messages with different injection rates – MPI v.s. LCI with/without the send immediate optimizations.

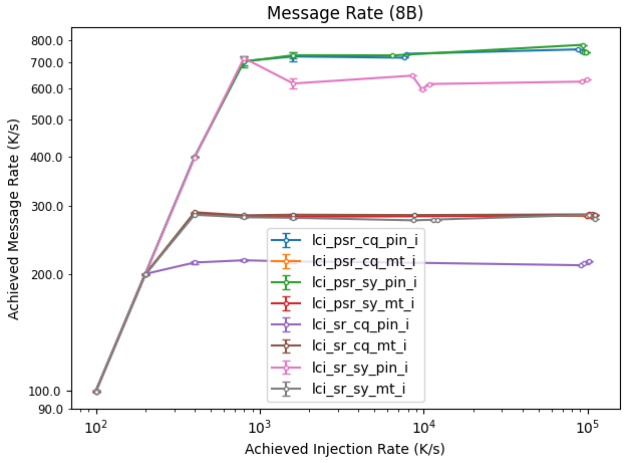

Figure 2: Achieved message rate of 8B messages with different injection rates – Different LCI configurations (with the send immediate optimizations).

Fig. 1 and Fig. 2 show the achieved message rate of 8B messages with attempted injection rates ranging from 100K/s to 1600K/s and unlimited. We set the batch size to 100 and the total number of

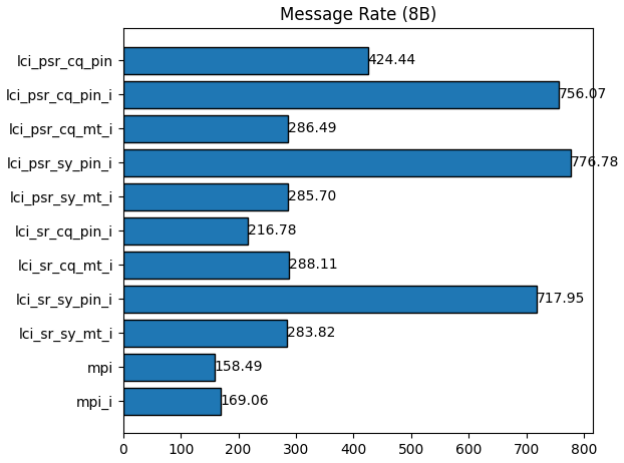

Figure 3: The highest achieved message rate of 8B messages across all injection rates.

messages to 500K. (The numbers are chosen so that the system will not run out of memory and crash.) For ease of comparison, we also add Fig. 3, which shows the highest achieved message rate across all injection rates.

With 8B messages, each parcel is transferred through one message: the header message; all those messages have the same tag. We have the following observations:

- For almost all cases, the achieved message rate first matches the injection rate almost perfectly and then plateaus. The only exception is the achieved message rate of the MPI parcelport without the send immediate optimization (*mpi*): it first increases and then decreases. It shows some component of the MPI parcelport data path is not stable under high message injection pressure.
- Comparing *lci_psr_cq_pin_i* and *lci_psr_cq_mt_i*, we see that a dedicated progress thread improves the message rate by up to 2.6x. Profiling results show the difference mainly comes from thread contention. When multiple worker threads call the progress function, they contend on various resources such as the network receive queue and completion queue, the matching table, the LCI completion queue, and some internal counters. Even though we carefully designed the thread-safe version of the LCI progress function using fine-grained try-locks and atomic variables, thread contention in the progress engine still makes a great difference when the incoming message rate is high, and has a bigger impact than the completion type and communication primitive, as all the *mt_i* variants are stuck at around 285K/s.
- Comparing *lci_psr_cq_pin_i*, *lci_sr_cq_pin_i*, we can see the performance impact of communication primitives. Using two-sided receives rather than one-sided put reduces the message rate by up to 3.5x. Using two-sided communication adds the overhead of posting receives and matching sends to receives. This causes additional load on the progress engine, as it has to handle the receive of unexpected messages. Second, the achieved message rate is determined by both the rate of posting receives and the

rate of polling the completion objects, and balancing these two is difficult.
- Without the send immediate optimizations, all the LCI parcelport variants achieve message rates of around 400K/s (we show only one of these variants in this paper, to avoid clutter). It means aggregation (the lack of the send immediate optimization) yields mixed results for the LCI parcelport. For some variants such as *lci_psr_cq_pin*, removing the aggregation (*lci_psr_cq_pin_i*) improves the message rate by up to 80%. For other variants such as *lci_sr_cq_pin* (not shown in the figure, but achieves around 400K/s), removing the aggregation (*lci_sr_cq_pin_i*) reduces the message rate by half. It shows aggregation can help the performance by reducing the small-message injection pressure if the underlying network stack is not efficient enough to handle it, but will hurt the performance otherwise as it adds additional software overhead and can become a bottleneck itself.

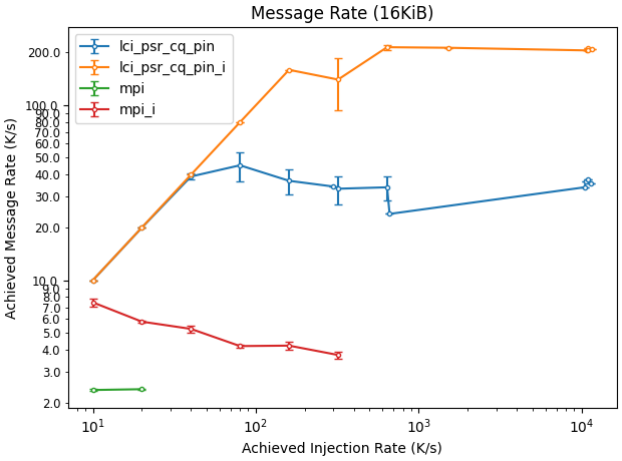

Figure 4: Achieved message rate of 16KiB messages with different injection rates – MPI v.s. LCI with/without the send immediate optimizations.

Fig. 4 and Fig. 5 show the achieved message rate for 16KiB messages with attempted injection rates ranging from 10K/s to 640K/s and unlimited. We set the batch size to 10 and the total number of messages to 100K. The standard deviations of some data points are too large so we shrink the error bar by 2.49x. Similarly, we add Fig. 6 to show the highest achieved message rate across all injection rates.

With 16KiB messages, each parcel is transferred through two messages: one header message and one follow-up message. Under high injection rates, the system will have a large number of messages transferred simultaneously, half with the same tag (the header messages) and half with different tags (the follow-up messages). We have the following observations:

- The LCI parcelport achieves up to 30x more throughput than the MPI parcelport. As the injection rate increases, the achieved message rates of both MPI parcelport variants keep decreasing. This shows it is very difficult for the MPI parcelport to receive

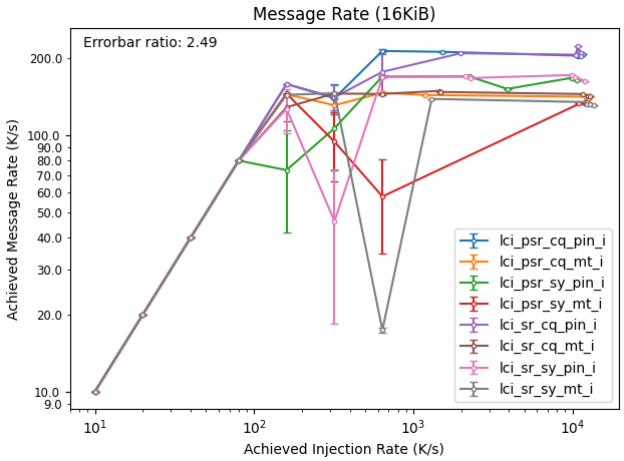

Figure 5: Achieved message rate of 16KiB messages with different injection rates – Different LCI configurations (with the send immediate optimizations). To reduce clutter, we shrink the error bar by 2.49x.

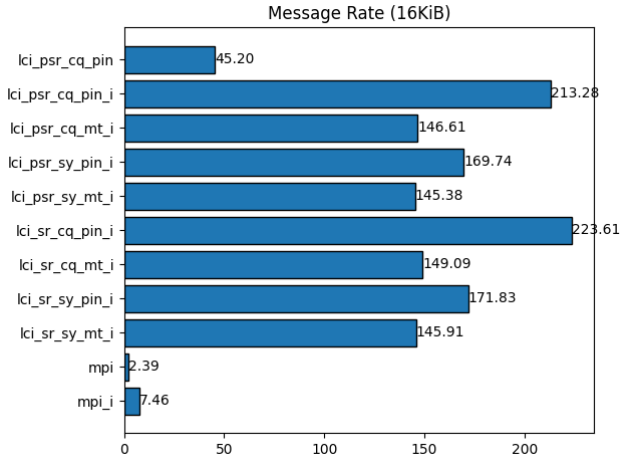

Figure 6: The highest achieved message rate of 16KiB messages across all injection rates.

a large number of concurrent messages with arbitrary source ranks and different tags.
- All the LCI parcelport variants with synchronizers exhibit large oscillations after reaching their peak message rate. Instead, those with completion queues stay stably at their peak message rate. While we have not identified the actual cause, one hypothesis is that these oscillations are due to a resonance between synchronizer polling rate and message arrival rate. In addition, using completion queues can improve the peak message rate by 25% - 30%.
- All the LCI parcelport variants with a dedicated progress thread achieve higher throughput than their counterparts using worker

threads calling progress function. The dedicated progress thread improves the message rate by 17% - 50%.
- All the LCI parcelport variants without the send immediate optimization achieve message rates between 40K/s and 50K/s. It shows the aggregation provided by the parcel queue cannot help the message rate of large messages. We attribute this to the fact that they cannot aggregate zero-copy chunks while suffering from the additional overhead of aggregation.

4.2 Latency

We use a multi-message ping-pong-like microbenchmark to study latency: a fixed number (determined by *window size*) of messages of a fixed size are sent back and forth between two processes for a fixed number (determined by *step number*) of iterations. Every "ping" and "pong" is performed by a different HPX task. Essentially, the task dependency graph consists of multiple chains of tasks that alternate between the two processes. *Window size* determines the number of chains and *step number* determines the length of the chains. We measure the total execution time and calculate the average time per iteration to be the one-way latency of the message.

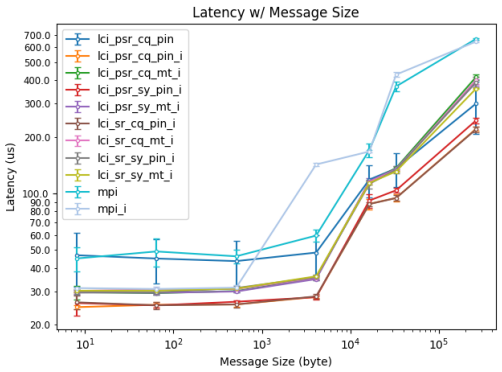

Figure 7: Single-message ping-pong latency with different message sizes.

Fig. 7 shows the message latency with different message sizes. We fix the window size to be 1. The HPX zero-copy serialization threshold is kept at 8192 bytes. We have the following observations:

- The baseline implementation of the LCI parcelport (*lci_psr_cq_pin*) always has lower latency than the MPI parcelport. The MPI parcelport with the send immediate optimization (*mpi_i*) performs reasonably well when the messages are smaller than 1KB: it is only 1.3x worse than *lci_psr_cq_pin*. However, it exhibits much worse latencies for larger messages (3-5x worse than *lci_psr_cq_pin*), potentially due to some protocols switch in the MPI/UCX layer.
- For all LCI parcelport variants, the send-immediate optimization always helps reduce the message latency. This is expected because (a) aggregation would not bring any benefits since there is only one message being transferred at any time (b) the send-immediate optimization bypasses the connection cache and parcel queues and thus reduces the software overhead.
- The LCI parcelport variants with the send immediate can be roughly divided into two groups. All the variants with a dedicated progress thread and using completion queues yield the lowest latency. (Note that *psr_sy* also uses a completion queue for the header message on the receiver side due to the implementation limitation of the current LCI put.) The reason is two-fold. First, having a dedicated progress thread, compared to having all the worker threads calling into the progress engine when idle, guarantees fast and steady communication responses. Second, using a dedicated progress thread and completion queue has a faster code path with fewer locks and atomic operations.

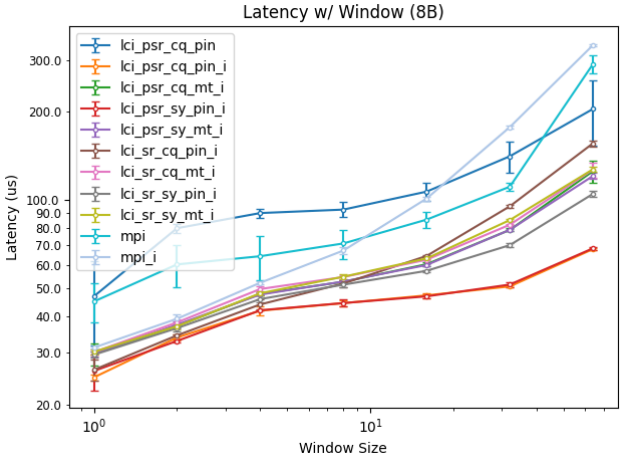

Figure 8: Latency of 8B messages with different window sizes.

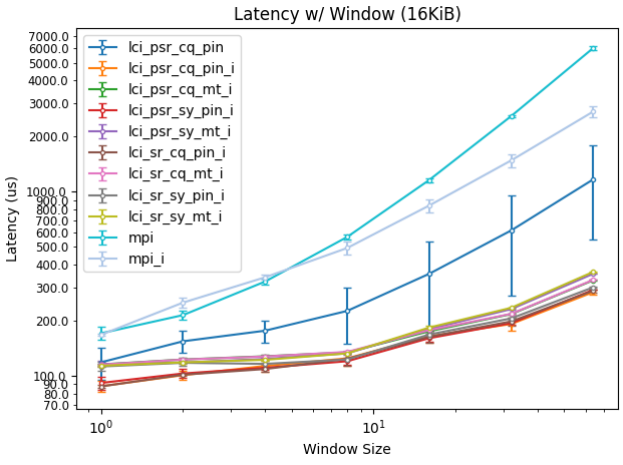

Figure 9: Latency of 16KiB messages with different window sizes.

Table 3: Rostam System Configuration.

CPU	Intel(R) Xeon(R) Gold 6148 CPU (Skylake) (2 sockets, 40 cores per node)
Memory	96 GB, DDR4
Storage	1TB Local NVMe SSD
NIC	Mellanox ConnectX-3
Interconnect	FDR InfiniBand (4x14Gbps)
Max Allowed Nodes/Job	16 Nodes
OS	Red Hat Linux 8.8
Compiler	GCC 10.3.1
Software	OpenMPI 4.1.5, UCX 1.14.0

Fig. 8 and Fig. 9 show the 8B and 16KiB message latency with varying window sizes from 1 to 64. The intention of these experiments is to study the ability of the parcelport to overlap the communication time of different messages. We have the following observations:

- In all cases, the latency increases with the window size. This shows a general trend that more concurrent messages lead to larger software overhead.
- For large messages, the latency gap between the MPI and LCI parcelport becomes larger as the window size increases. The latency ratio between *mpi_i* and *lci_psr_cq_pin_i* increases from 2x (window size 1) to 9.6x (window size 64). This means MPI has difficulty dealing with a large number of concurrent messages.
- The send immediate optimization always helps reduce latency for the LCI parcelport, but exhibits mixed results for the MPI parcelport. Especially for small messages, *mpi_i* initially performs much better than *mpi*, but becomes worse as the window size grows. The switching point is window size 8. This shows again MPI performs poorly when handling a large number of concurrent messages.
- *lci_psr_cq_pin_i* is the best variant in almost all cases, which is expected. All the other variants become worse when there are more concurrent small messages, potentially due to the locks and atomic operations in their message-passing code path. Posting receives will involve more contention in the matching table and the producer side of completion objects. Worker threads making progress will involve more contention on the progress engine. Synchronizer will involve contention on the synchronizer pools. The performance impact of the progress type and the communication primitives is larger than the impact of the completion type.

5 APPLICATION BENCHMARK

To understand how the performance difference between the MPI parcelport and the LCI parcelport in the microbenchmarks translates into application speedup, we use Octo-Tiger [21] as an application-level benchmark. Octo-Tiger is an astrophysics application simulating the evolution of binary star systems based on the fast multipole method on adaptive octrees. It is built on top of HPX and uses HPX actions and local control objects extensively to achieve asynchronous execution and computation-communication overlap.

Experiments in this section are performed on SDSC Expanse and Rostam. Table.2 and Table.3 shows their system configuration. These two platforms use different processors and different generations of InfiniBand, so we use them to study how the LCI parcelport performs with different hardware. Octo-Tiger has a configuration parameter that determines the maximum level of the adaptive octtree, which in turn determines the total number of tasks. Octo-Tiger uses space-filling curves to partition the tree nodes into processes. Increasing the number of levels increases computation and intra-process communications faster than inter-process communications. We set the number of levels to a relatively small number (6 on SDSC Expanse and 5 on Rostam) to study configurations where inter-process communication is a significant bottleneck, as happens in strong scaling. We set the *stop step* (iteration count) to be 5 to get a reasonable total execution time. Each experiment is performed at least five times. We show the average and standard deviation in the figures. In all figures, *mpi_i* means the MPI parcelport with the send immediate optimization. *mpi* means the MPI parcelport without the send immediate optimization. The solid line with the left axis shows the absolute performance (step count per second). The hash line with the right axis shows the relative speedup of Octo-Tiger with the LCI parcelport compared to the two MPI variants. We use the same configuration of the LCI parcelport described in Sec.3.2.1 a.k.a *lci_psr_cq_rp_i*. We do not show all the other LCI parcelport variants because the difference between them is very small at the application level.

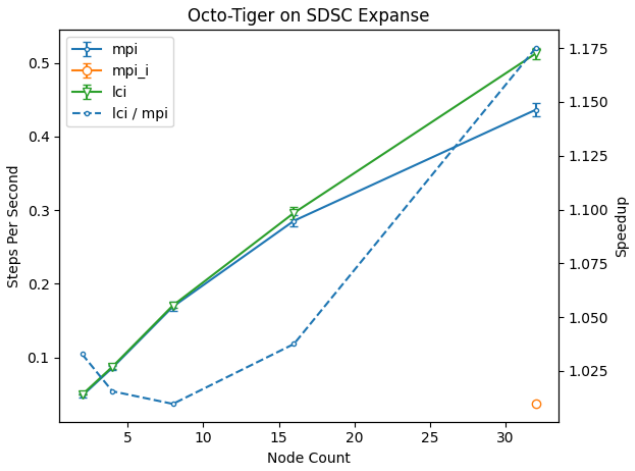

Figure 10: Step count per second of Octo-Tiger on SDSC Expanse with different numbers of computation nodes (strong scaling). *mpi_i* means the MPI parcelport with the send immediate optimization. *mpi* means the MPI parcelport without the send immediate optimization. *lci* represents the default configuration a.k.a *lci_psr_cq_rp_i*.

Fig. 10 and Fig. 11 show the step count per second of running Octo-Tiger with different numbers of computation nodes on SDSC Expanse and Rostam. We only show the 32-node result of running Octo-Tiger with *mpi* on SDSC Expanse since the *mpi_i* version is extremely slow. On both platforms, the LCI parcelport performs

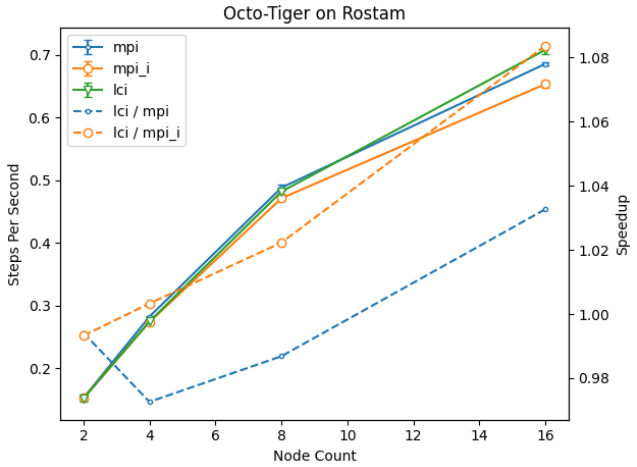

Figure 11: Step count per second of Octo-Tiger on Rostam with different numbers of computation nodes (strong scaling). *mpi_i* means the MPI parcelport with the send immediate optimization. *mpi* means the MPI parcelport without the send immediate optimization. *lci* represents the default configuration a.k.a *lci_psr_cq_rp_i*.

better as the node count increases. Compared to *mpi_i*, the MPI parcelport with send immediate optimization, we get up to 13.6x speedup on SDSC Expanse and 1.08x speedup on Rostam. Compared to *mpi*, the original MPI parcelport, we get up to 1.175x speedup on SDSC Expanse and 1.04x speedup on Rostam. The LCI parcelport does not get much speedup on Rostam potentially due to the limited node count. The comparison between *mpi* and *mpi_i* shows that aggregation helps the MPI parcelport on both platforms. *mpi_i* is very inefficient on SDSC Expanse, potentially due to its high CPU core count. Profiling results show that it spent the vast majority of time inside the *MPI_Test* function, spinning on the blocking lock of the *ucp_progress* function. This shows the importance of using atomic operations and/or fine-grained try-lock instead of coarse-grained blocking lock to ensure thread safety.

We also performed some experiments to study the different LCI parcelport configurations with Octo-Tiger. However, Octo-Tiger does not stress communication as hard as the microbenchmarks. We found that the performance differences between the different configurations were obscured by the background noise from the Octo-Tiger layer. As a result, we leave this study to future work.

6 RELATED WORK

There is a significant amount of research on Asynchronous Many-Task Systems in recent years. Some, such as the task constructs of OpenMP [2], Cilk [13], or Intel Threading Building Blocks [18], target shared memory multiprocessors and require an MPI+X programming model to expand beyond one node. On the other hand, there are task systems that have their own inter-node communication layers. Examples of such systems include PaRSEC [7], Legion [4], HPX [15, 16], DDDF [10], StarPU [1], and TaskTorrent [8]. There are differences in how much they expose the details of inter-node communication to users but users generally should not and need not make calls to external communication libraries. This simplifies programming and allows for better integration of the communication library with the scheduler. This paper is about HPX, a system in the second category, but MPI+task systems share many similar challenges.

There is a range of communication libraries and languages in the parallel computing area. MPI [23] has been the most popular one, well-supported on almost all platforms, and is usually the first choice when people start to develop new applications. UPC [12], UPC++ [3], and OpenSHMEM [9] focus on one-sided communication and the PGAS programming model, which provides users with an abstraction of shared memory across nodes. While the previous communication libraries/languages are intended to be directly used by applications, GASNet-EX [6], Libfabric [24], and UCX [27] are low-level communication libraries designed to support higher-level systems. GASNet-EX focuses on one-sided active messages and RDMA operations. It uses an *event* handler similar to *MPI_Request* to test/wait for nonblocking operations. Libfabric has both two-sided send-receive and one-sided RDMA operations and uses completion queues for completion notification. UCX has both two-sided send-receive, one-sided active message, and RDMA operations and uses callback functions for completion notification.

A few existing works study the communication layer of distributed task systems. HCMPI [10] provides an MPI wrapper to make the task scheduler aware of the communication jobs. It funnels all MPI calls to a dedicated communication thread due to the inefficiency of multithreaded MPI and polls a list of pending MPI requests using `MPI_Test` regularly. PaRSEC uses MPI and LCI as its communication backend. The MPI backend has been described in [26]. The MPI implementation also funnels most MPI calls to a dedicated communication thread. It uses *MPI_Testsome* on a small array of pending requests to mitigate the overhead of request polling. [26] proposes a new MPI functionality of attaching callbacks to MPI requests to further optimize the polling for pending communication. [22] studied the integration between PaRSEC and LCI. It uses a dedicated communication thread and a dedicated progress thread to handle the communication. [5] briefly describe the communication layers of Charm++ and focus on eliminating the memory copies of communication inside the Charm++ runtime.

Although this work and [22] share the same communication libraries, we work on different task systems with different communication layer abstractions, and thus the usage of LCI becomes different. We also use different microbenchmarks and benchmarks to evaluate the systems and design decisions. These two works should be seen as complementary and prove the generality of LCI.

7 CONCLUSION

We have studied microbenchmark and benchmark results in depth in the previous sections. In this section, we summarize the main lessons we learn from these results and discuss future work.

7.1 Main Lessons

The LCI parcelport outperforms the MPI parcelport in almost all cases, sometimes by a substantial proportion: It seems that (multithreaded) MPI has a difficult time when dealing with a large number of concurrent messages: Latency increases (Fig. 8 and Fig. 9)

and the message rate sharply decreases (Fig. 1 and Fig. 4). The most significant indication is provided by Fig. 10 where *mpi_i* becomes 13.6x slower when running Octo-Tiger on SDSC Expanse. Profiling results confirm that it is due to the use of coarse-grained blocking locks inside MPI/UCX layer.

Message aggregation yields mixed results: Our microbenchmarks show that aggregation helps with the communication performance when the underlying software stack cannot function efficiently under high stress of message injection, such as the case for *mpi*/*mpi_i* and *lci_sr_cq_pin*/*lci_sr_cq_pin_i*. The most notable benefit is that it could help *MPI* avoid abysmal performance when running Octo-Tiger on SDSC Expanse (Fig. 10). However, the benefit of less stress for the underlying network stack is balanced by the additional software overhead of aggregation and the balance depends on the communication characteristics. For example, aggregation cannot help the LCI parcelport for the message rate of large messages (Fig. 4) and latency (Fig. 7). This trade-off has also been observed by previous research [25].

Using a dedicated progress thread is almost always helpful: Having multiple threads making progress will lead to more thread contention and cache misses in the progress engine and a less stable response to the network events. Microbenchmark results show that it will constrain the 8B message rate to around 285K/s and the 16KiB message rate to around 150K/s. A dedicated progress thread alleviates the bottleneck by 2.6x (Fig. 3) and 1.5x (Fig. 6) respectively. It also helps with the message latency (Fig. 7).

Polling one completion queue is preferable to polling multiple requests or synchronizers: Request objects are useful when users want to synchronize individual operations. However, for applications such as task systems that are event-driven and do not care about completion ordering, polling one completion queue leads to fewer CPU cycles and less thread contention than polling a pool of individual requests. In the 16Kib message rate microbenchmark, request pools constrain the message rate to 170K/s and completion queues lift the bar by 30% (Fig. 6). All the variants with completion queues also exhibit a much smoother line than their counterparts with request pool (Fig. 5). It does not have effects on other microbenchmarks because they do not have a large number of pending requests.

A put with a remote completion (queue) signal achieves better performance than send-recv at high short-message rates: This is shown by the 3.5x performance gap for the 8B message rate in Fig. 2 between *lci_psr_cq_pin_i*, and *lci_sr_cq_pin_i*. A possible reason is the advantage of not posting receives and matching sends to receives, thus reducing the thread contention on the progress engine. However, in many other cases, variants with the put show similar performance compared to their send-recv counterpart. The use of puts also simplifies programming.

7.2 Future Work

The multithreaded performance of the LCI parcelport is far from optimal. Modern Mellanox NICs can reach a peak message rate of more than 100 Mpps [17]. This should be translated to at least tens of millions of parcels per second for the HPX parcelport layer. However, the current LCI parcelport barely reaches 750K/s. We attribute a large part of this gap to the contention on low-level network resources. Currently, the LCI parcelport only uses one LCI device per process which maps to one low-level network context per process. This causes severe thread contention when the sender injects messages into the network. Previous work [29][30][19][14] has shown that replicating low-level network resources could greatly increase message rates at the network microbenchmark level. However, it is tricky to correctly and efficiently use multiple instances of network resources for a task system. We plan to pursue this important research topic in our future work.

We can roughly classify the performance improvements with LCI into two categories: (a) The performance difference between *lci_psr_cq_pin_i* and *lci_sr_sy_mt* shows the contributions from the HPX parcelport layer enabled by the more flexible and versatile LCI API. (b) The performance difference between *lci_sr_sy_mt_i* and *mpi_i* shows the contributions from the communication library layer due to the better multithreaded performance of LCI. We mainly covered the parcelport layer improvements in this paper. There are many important design decisions in the LCI layer that affects the task system performances that we have not discussed here due to page limitations.

It would be nice to compare LCI not only with MPI but also with other communication libraries. Other HPX parcelports are currently under development and we shall be able to pursue this direction once the development completes. Since many of the OpenMPI communication calls that are used by the MPI parcelport directly call the corresponding UCX functions with only a thin layer, we think that part of the performance gap between the MPI parcelport and the LCI one will persist if MPI is replaced by UCX, with no further design changes.

ACKNOWLEDGMENTS

The authors would like to thank Patrick Diehl and Gregor Daiss for their advice on compiling and running Octo-Tiger and the Center of Computation and Technology at Louisiana State University for providing access to its Rostam cluster. This research was supported in part by NSF grants 1908144 and 1909015. This work used the Expanse system at the San Diego Supercomputer Center through ACCESS allocation CCR130058.

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://doi.org/10.1002/cpe.1631

[2] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (2009), 404–418. https://doi.org/10.1109/TPDS.2008.105

[3] 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. https://doi.org/10.1109/IPDPS.2019.00104

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*. IEEE/ACM, 1–11. https://doi.org/10.1109/SC.2012.71

[5] Nitin Bhat, Sam White, and Laxmikant V. Kale. 2022. Enabling Support for Zero Copy Semantics in an Asynchronous Task-Based Programming Model. In *Euro-Par 2021: Parallel Processing Workshops*, Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl,

Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci (Eds.). Springer International Publishing, Cham, 496–505.

[6] 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. https://doi.org/10.1007/978-3-030-34627-0_11

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

[8] Léopold Cambier, Yizhou Qian, and Eric Darve. 2020. TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++. , 16-26 pages. https://doi.org/10.1109/PAWATM51920.2020.00007

[9] 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. https://doi.org/10.1145/2020373.2020375

[10] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating Asynchronous Task Parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 712–725. https://doi.org/10.1109/IPDPS.2013.78

[11] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. 2018. A Lightweight Communication Runtime for Distributed Graph Analytics. In *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. IEEE, 980–989. https://doi.org/10.1109/IPDPS.2018.00107

[12] 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. https://doi.org/10.1145/1188455.1188483

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[14] 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. https://doi.org/10.1145/2597652.2597657

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

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

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[18] Wooyoung Kim and Michael Voss. 2011. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Software* 28, 1 (2011), 23–31. https://doi.org/10.1109/MS.2011.12

[19] 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. https://doi.org/10.1109/PAW-ATM49560.2019.00010

[20] Patrick MacArthur, Qian Liu, Robert D. Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. 2017. An Integrated Tutorial on InfiniBand, Verbs, and MPI. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2894–2926. https://doi.org/10.1109/COMST.2017.2746083

[21] 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 (04 2021), 5345–5382. https://doi.org/10.1093/mnras/stab937 arXiv:https://academic.oup.com/mnras/article-pdf/504/4/5345/37975469/stab937.pdf

[22] 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* (Salt Lake City, UT, USA) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/3605573.3605642

[23] MPI Forum. 1993. MPI: a message passing interface. In *1993 ACM/IEEE Conference on Supercomputing (SC93)*. ACM, 878–883. https://doi.org/10.1145/169627.169855

[24] OFI Working Group (OFIWG). 2023. Libfabric Programmer's Manual.

[25] C.D. Pham and C. Albrecht. 1999. Optimizing message aggregation for parallel simulation on high performance clusters. In *MASCOTS '99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 76–83. https://doi.org/10.1109/MASCOT.1999.805042

[26] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. 2021. Callback-based completion notification using MPI Continuations. *Parallel Comput.* 106 (2021), 102793. https://doi.org/10.1016/j.parco.2021.102793

[27] 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. https://doi.org/10.1109/HOTI.2015.13

[28] Marc Snir, Hoang-Vu Dang, Omri Mor, and Jiakun Yan. 2023. LCI: A Lightweight Communication Interface v1.7. https://github.com/uiuc-hpc/LC/blob/icpp23/doc/LCI.pdf

[29] 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)*. 803–812. https://doi.org/10.1109/PADSW.2018.8645059

[30] 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. https://doi.org/10.1145/3555819.3555820