Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine

Omri Mor
University of Illinois
Urbana–Champaign, USA
omrimor2@illinois.edu

George Bosilca
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
bosilca@icl.utk.edu

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

ABSTRACT

There is a growing interest in Asynchronous Many-Task (AMT) runtimes as an efficient way to map irregular and dynamic parallel applications onto heterogeneous computing resources. In this work, we show that AMTs nonetheless struggle with communication bottlenecks when scaling computations strongly and that the design of commonly-used communication libraries such as MPI contribute to these bottlenecks. We replace MPI with LCI, a Lightweight Communication Interface that is designed for dynamic, asynchronous frameworks, as the communication layer for the PaRSEC runtime. The result is a significant reduction of end-to-end latency in communication microbenchmarks and a reduction of overall time-to-solution by up to 12% in HiCMA, a tile-based low-rank Cholesky factorization package.

CCS CONCEPTS

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

KEYWORDS

message-passing, MPI, lightweight communication, asynchronous many-task, dynamic runtime, low-rank Cholesky, strong scaling

ACM Reference Format:
Omri Mor, George Bosilca, and Marc Snir. 2023. Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605642

1 INTRODUCTION

The search for more efficient hardware is leading to increasingly complex and heterogeneous platforms. The search for more efficient algorithms is resulting in codes that are more irregular and adaptive. The traditional bulk-synchronous computation models are not well suited to either, as they force unnecessary synchronization. These developments have led to a resurgence of research into dynamic runtimes that can increase asynchrony and automatically manage the overlap of data movement with fine-grained computation. Their typical communication patterns exhibit these properties:

- Large numbers of independent messages sent and received simultaneously: a node may run hundreds of independent tasks, each producing and consuming data.
- Dynamically changing communication patterns.
- Separation between the control and data layers: control messages are generated and consumed by the runtime, while data messages are generated and consumed by computing tasks; they have different characteristics.
- Significant variability in message size: control messages are typically a fraction of the size of data messages, while data messages' size depends on task type and, possibly, input values.

Communication libraries such as MPI are designed and optimized for regular communications with infrequent long messages and do not support well the unique communication characteristics of dynamic runtimes: communication overheads become a bottleneck, especially when strong scaling, as the number of tasks per core and/or their size shrinks, so that communications cannot always be covered and large latencies result in idle computation resources. Performance can be improved by using a communication layer that is better adapted to the needs of these runtimes. In this work, we make the following contributions:

(1) Establish why strong scaling is challenging even when an application seems to expose sufficient parallelism.
(2) Describe the (new) communication abstraction of PaRSEC [4, 5] and how this abstraction is implemented with MPI.
(3) Describe LCI [11], the Lightweight Communication Interface, and how we use it to implement PaRSEC communication.
(4) Demonstrate sustained performance uplift over MPI in a series of PaRSEC microbenchmarks.
(5) Show how this translates to improved real performance in HiCMA [6–8], a state-of-the-art tile-based low-rank Cholesky factorization application that uses PaRSEC.

The rest of this paper is organized as follows. In Section 2 we discuss the communication needs of dynamic runtimes and issues that arise with strong scaling. Section 3 describes related work. Section 4 describes the new PaRSEC communication engine and its current MPI backend. In Section 5 we describe LCI, why it is a better fit for dynamic runtimes, and how we utilize its feature set in PaRSEC. We demonstrate our experimental results comparing the MPI and LCI backends using both microbenchmarks and HiCMA in Section 6. We conclude with future research directions in Section 7.

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605642

2 BACKGROUND

2.1 Communication in Dynamic Runtimes

At a fundamental level, a parallel execution consists of a partially ordered set of operations; the partial *program order* represents data and control dependencies among operations. Specifying a parallel computation at this level is not practical for high-level programming and does not provide an efficient execution model due to scheduling and communication overheads. A practical model will use coarse-grain "operations" and will aggregate individual communications and synchronizations. This coarsening imposes unnecessary restrictions on the execution order of operations, which may result in a loss of parallelism. Different programming models choose different compromises between these two conflicting forces. At one extreme, the *Bulk-Synchronous Parallel (BSP)* model focuses on aggregation. In this model, computations involve a fixed number of processes iterating through the same sequence of epochs. There are no dependencies between operations executed in the same epoch by different processes. This model avoids scheduling overheads and facilitates the aggregation of all interprocess communications, epoch by epoch. On the other hand, it requires that the computation be decomposed into a fixed number of repeated "tasks" (the execution of an epoch by a process), all of the same size. For regular algorithms that are amenable to such a decomposition, the BSP model reduces runtime overheads. Communications will typically involve either a few large messages per epoch or collective operations. MPI was designed at a time when many HPC applications were bulk-synchronous, so MPI implementations have tended to optimize for such use cases. At the other extreme, *Asynchronous Many-Task (AMT)* models represent computations as a set of tasks and dependencies between tasks. Tasks are scheduled dynamically when all their dependencies are satisfied and computation resources are available. This provides much more flexibility to express algorithms that are irregular, but imposes more work on the runtime. The runtime overhead is reduced by ensuring that tasks are large enough to amortize the (presumably constant) scheduling overhead; communication overheads are hidden by *over-decomposing*, i.e., having more tasks than compute resources, so that a ready task can execute while other tasks wait for dependencies to be resolved.

Runtimes that implement AMT models typically use two classes of messages: control messages that tend to be rather short—on the order of a few kilobytes at most—and data messages that handle data dependencies across tasks. These tend to be longer than the control messages, but are shorter than in the BSP model, as over-decomposition reduces task data size. It is common for these runtimes to have a notion of task priority, allowing the scheduler to prioritize the execution of tasks on the critical path. It is also necessary to prioritize communications in such a manner that high-priority tasks receive data before low-priority tasks. In addition, control messages should not be delayed by data messages.

Task graphs tend to be complex and may be data-dependent; task execution and communication times vary. Therefore, it is usually not possible to predict message arrival order, and the runtime processes the incoming messages in the order of their arrival. As task execution is dependent on communications, this nondeterminism inherently leads to nondeterministic task execution order and thus to the order of outgoing communications that are subsequently generated. Thus, most applications using dynamic runtimes exhibit non-deterministic communication patterns [9]. Different communication orders may have a significant impact on the order in which the task graph is executed.

2.2 Strong Scaling

When strong scaling a problem to larger core counts, it becomes necessary to increase the total number of tasks by reducing their individual size. The amount of computation and communication per processor will decrease, but the relative amount of runtime overhead will increase since the (presumably fixed) overhead of task scheduling will not be amortized by the shorter computation. In addition, messages will be shorter, so that message rate, rather than bandwidth, becomes the critical communication bottleneck. If over-decomposition is reduced in order to palliate these effects, then message latency and proper handling of priorities become critical, as communication may not always be covered by computation.

3 RELATED WORK

Communication libraries and runtimes have been designed for a variety of purposes and with different resulting characteristics. Low-level libraries such as IB verbs target the InfiniBand hardware abstraction directly; while this permits an application complete control over the desired behavior, it is difficult to utilize and port to different network hardware, so few high-level runtimes implement communications directly over these abstractions. A notable exception to this is Charm++ [20], which has direct support for IB verbs and other low-level communication interfaces [21, 31].

Thus for portability and familiarity, most dynamic runtimes have used widespread interfaces such as MPI [22], thereby reducing the effort of porting to new systems and allowing the runtime developers to focus on core concerns. PaRSEC [4, 5] and HPX [19] both primarily use MPI, while Charm++ uses MPI on networks that do not yet have direct support. Similarly, Legion [1] uses GASNet [2] to support its active messaging paradigm. These interfaces have traditionally been used for other purposes, and while they have recently gained support for features that better support dynamic runtimes–for instance, the relaxed ordering requirements in MPI 4.0 [23] and the continuations proposal [26]—both encounter the difficulties described in Section 2.1.

More recently, communication libraries such as the Open Fabrics Interfaces (OFI, also called libfabric) [17] and Unified Communication X (UCX) [28] have emerged as a middle ground. These thread between the low-level hardware interfaces and application-oriented APIs, providing a performance portability layer for communication. Both GASNet and MPI implementations have used these middleware in lieu of their own low-level backends, but they have also been successfully used directly by dynamic runtimes [10].

LCI [30] is comparable to OFI or UCX in its goals of portable performance. It is a small, easily modifiable research library that has been used to study API design and implementation issues in the context of heavily contented multithreaded communication [12, 13] and graph applications [11, 14, 15].

Listing 1: PaRSEC communication engine API

```
typedef int (*am_cb_t)(comm_engine_t *ce, tag_t tag,
        ↪ void *msg, size_t msg_size, int src, void
        ↪ *cb_data);
typedef int (*onesided_cb_t)(comm_engine_t *ce,
        ↪ mem_reg_t lreg, ptrdiff_t ldispl,
        ↪ mem_reg_t rreg, ptrdiff_t rdispl, size_t
        ↪ size, int remote, void *cb_data);
int tag_reg(tag_t tag, am_cb_t cb, void *cb_data,
        ↪ size_t len);
int mem_reg(void *mem, size_t count, datatype_t dtype,
        ↪ mem_reg_t *lreg, size_t *lreg_size);
int send_am(comm_engine_t *ce, tag_t tag, int remote,
        ↪ void *addr, size_t size);
int put(comm_engine_t *ce, mem_reg_t *lreg, ptrdiff_t
        ↪ ldispl, mem_reg_t rreg, ptrdiff_t rdispl,
        ↪ size_t size, int remote, onesided_cb_t
        ↪ l_cb, void *l_cb_data, tag_t r_tag, void
        ↪ *r_cb_data, size_t r_cb_data_size);
int progress(comm_engine_t *ce);
```

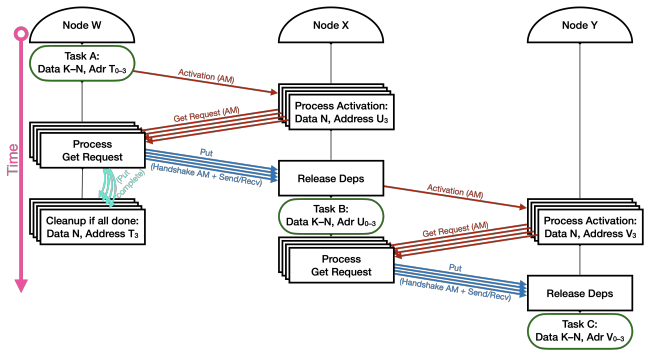


Figure 1: PaRSEC communication example. Task A runs on node W with descendant tasks B and C on nodes X and Y, respectively. There are four dataflows that must be propagated as part of the broadcast.

4 PARSEC COMMUNICATION

4.1 The PaRSEC Communication Engine

PaRSEC recently integrated a new communication model and API, shown in Listing 1, that abstracts some of the underlying implementation details and lets the communication layer work independently of the backend used. An active-message paradigm was chosen as a good fit for the asynchronous nature of PaRSEC.

As part of initialization, PaRSEC registers two active messages:

ACTIVATE: inform that a task completed, activating descendants
GET DATA: retrieve data associated with a completed task

Usage of these two active messages and the `put` function is demonstrated by Figure 1. When a task completes, it is determined where its descendants are to be executed and ACTIVATE messages sent to those destinations. Upon receipt of the ACTIVATE message, the process will evaluate the relative priority of successor tasks depending on data from the predecessor and use these priorities to determine whether to request data immediately or defer it by giving precedence to other communications. The prioritization is done independently for each output data of the completed task. Once the communication has a sufficiently high relative priority, a GET DATA message is sent to the process on which the predecessor task was executed, which will subsequently begin a `put` operation.

4.2 MPI Backend

The MPI backend for PaRSEC uses a structure to encapsulate callbacks for communications, including information such as what function to call and what arguments are to be supplied. An array of these structures is paired with a parallel array of `MPI_Request` for communications currently in progress.

4.2.1 Active Messages. Active message handling is split into two parts—`tag_reg` registers an active message tag with the communication runtime while `send_am` sends an active message to a destination using a registered tag.

Messages are received using MPI persistent receives: a static number—configured as five per tag in the current implementation—of requests are initialized (`MPI_Recv_init`) and started (`MPI_Start`) during tag registration. Since the order and origin of communications is determined dynamically at execution time, the wildcard sender `MPI_ANY_SOURCE` is used. Wildcard tags are *not* needed, as all active message tags are explicitly registered and the tag implicitly determines which callback to invoke by dint of the indices of completed requests in the parallel array structure. After a message is received and its registered callback is executed, the request is re-enabled so that further messages can be received.

Active message sizes typically fall within the range where MPI implementations will use an "eager" communication protocol to transport the data. As such, a blocking `MPI_Send` is used with the tag registered for the selected active message as the MPI tag.

4.2.2 Data Transport. While the high-level API uses a one-sided communication scheme for bulk data transport, the MPI backend does not use MPI RMA interfaces to implement these functions. The PaRSEC `put` interface requires remote completion notifications, which is not supported by standard MPI RMA. Additionally, the asynchronous nature of the PaRSEC runtime would likely necessitate using dynamic RMA windows that attach and detach memory frequently, which are known to have performance limitations under most circumstances [25]. It is certainly not impossible to use the MPI RMA interfaces to implement the PaRSEC `put` API, but exploring this option has been left for future work.

Thus bulk data transport in the MPI backend is implemented using non-blocking two-sided communication APIs in conjunction with a handshake message. The origin process of the `put` sends an active message informing the target process that it will send data on a specified tag. This handshake includes additional information such as where to receive the data, how much data will be sent, and what callback to call for completion of the `put`. The active message handshake callback at the target will post a receive for the specified buffer with the requested tag; as long as the source does not reuse the same tag before the communication is complete, this will not

conflict with any other communication. Since MPI message ordering semantics are not required for correct operation in PaRSEC, they are disabled using the `mpi_assert_allow_overtaking` Info key if this is supported by the MPI implementation.

A maximum of 30 data transfers—both sends and receives—are allowed to be actively polled concurrently. If there is insufficient space in the global array when a `put` is started, posting the send is deferred until prior communications are complete. If, when a handshake message arrives, there is insufficient space, an `MPI_Request` is allocated dynamically from a memory pool and the receive posted using this request; however, requests are only explicitly polled for progress when in the global array, so this dynamic request will be polled only when prior communications complete and the request promoted to the global array. This mechanism is intended to prevent communication bandwidth from being split across too many concurrent communications. This may reduce aggregate bandwidth, but also reduces the average completion time of individual communications. This was deemed an acceptable trade-off when scaling: tasks become ready faster, potentially reducing idle time.

4.2.3 Progress. As implied by the above description of active messages and data transport, the global array of `MPI_Request` is of length $5 \times N_{am} + 30$, where N_{am} is the number of registered active message tags. To allow overlap between progress on existing communications and starting new ones, the global request array is polled using `MPI_Testsome`. For each completed communication, the corresponding callback is fetched from the parallel array of callbacks and executed using the supplied arguments. After the callbacks for all completed communications are executed, incomplete communications are moved forward in the global array so that the free entries are always at the back. While there is free space in the array, deferred sends are started and dynamic receives promoted in FIFO order. If no communications were completed by `MPI_Testsome`, the `progress` function returns; otherwise, it repeats, attempting to make further communication progress.

4.3 The PaRSEC Communication Thread

PaRSEC utilizes a communication thread to handle communication initialization, progress, and completion. The thread can be configured to execute on a dedicated computing resource or be allowed to "float", stealing time from compute threads; which strategy is preferred is hardware-dependent. This dedicated thread is used to alleviate performance limitations on the part of MPI implementations. Prior work [24] has demonstrated that MPI performance can decrease significantly when many threads communicate concurrently. PaRSEC allows computing threads to send Activate messages directly rather than funneling them through the communication thread, but as we show in Section 6, doing so can decrease performance and is therefore not the default.

The communication thread has four primary responsibilities:

(1) Aggregate Activate messages sent to the same destination
(2) Poll the communication engine `progress` function
(3) Send deferred Get Data messages
(4) Initiate deferred `put` communications

As discussed above, the `progress` function in the MPI backend both polls for any completed communications and executes associated completion callbacks. These callbacks can themselves start new communications, as shown in Figure 1. If these communications could not be immediately started, they are added to "deferred" queues to be initiated later.

As a consequence of the communication thread design, these roles can interfere with each-other. For example, certain completion callbacks can take a long time to execute: an Activate message must unpack each aggregated activation, iterate over all local descendants of the task in question, determine which data are needed from the predecessor, and send Get Data messages as necessary. During the period of time that this callback is being executed, the communication thread cannot progress any other communication by e.g. matching an incoming message to a posted receive, preparing additional Activate messages to be processed, etc. Thus the latency of communications is greater than might otherwise be possible.

5 REDUCING COMMUNICATION LATENCY

5.1 Lightweight Communication Interface

The Lightweight Communication Interface [30] is designed explicitly to be consumed by libraries and frameworks, rather than by applications directly. It has been influenced by features provided by the InfiniBand verbs interface and mid-level communication libraries such as OFI [17] and UCX [28]. The primary goal is to be lightweight and support a reasonably high-level interface with features that can be supported directly by underlying communication hardware while *not* providing features that are better implemented by the high-level runtimes we target, such as datatype or serialization support. LCI is intended to, as directly as possible, support the communication requirements of dynamic runtimes, so as to avoid multiple layers of abstraction. Direct control of polling and progress is provided, as is efficient support for large numbers of concurrent threads and communications.

Communication calls in LCI are non-blocking: they either succeed or return a failure code indicating that there are insufficient resources to execute the requested operation and that the caller must progress existing communications before resubmitting the request. This allows the communication library to exert back-pressure on the application runtime, ensuring that it does not become overloaded. Communication completion can be signaled directly via a synchronizer object—roughly analogous to an MPI request in that it can be tested or waited on for completion—or dynamically with a completion queue or handler function. Execution runtimes using lightweight threads [27, 32] may opt to use synchronizers to easily unblock threads waiting on communications [12], while those using a centralized communication mechanism may prefer to poll completion queues or use an active message handler. LCI exposes three communication protocols for use by consuming runtimes:

Immediate short messages about the size of a cache line that can be sent inline from the user buffer
Buffered medium messages consisting of a few pages that are copied to pre-registered internal buffers
Direct long messages of any length that are sent via RDMA, using a rendezvous protocol if necessary

These are similar to protocols available in GASNet [2] or UCX [28] and allow the runtime to choose the appropriate size—for instance, a control message could be sent using the Immediate or Buffered

protocols, while data could be sent using Buffered or Direct protocols. Needless to say, which is appropriate for what use case depends on the runtime, and may be dynamic in nature.

5.2 Latency Reduction Techniques

There are several features available in LCI that fit the needs of the PaRSEC communication infrastructure and reduce communication latency or contention. The ability to dynamically respond to incoming messages via completion queues or handlers alleviates the need to poll individual communications as is necessary in the MPI backend. This fits in well with the design of the communication thread in PaRSEC, which can poll for any completed communications and handle them dynamically rather than needing to test each one in an array. The explicit progress call also avoids a source of congestion seen in the MPI backend where long active message callbacks prevent progress on communications. By putting the progress call on a separate thread or by judicious insertion of progress calls inside active message callbacks, we can prevent this undesired behavior.

Explicit protocol selection also lets PaRSEC make the correct choice of what protocol to use for Activate and Get Data messages versus what to use for the `put` operation. Additionally, LCI lets the user specify that buffers are dynamically allocated at the receiver; this is particularly useful for active messages, avoiding the need to probe for incoming messages followed by an allocation and explicit receive, or using a limited number of persistent receives that are re-enabled after completion.

5.3 LCI Backend for PaRSEC

5.3.1 Communication Progress. In contrast to the MPI backend, the LCI backend divorces progress on existing communications from executing the callbacks for active messages and starting new communications. This is possible due to the explicit progress call provided by LCI. In addition to the communication thread described in Section 4.3, the LCI backend spawns what we term a "progress thread" which is dedicated to calling the `LCI_progress` function. This thread is responsible for draining hardware completion queues, matching incoming messages with posted receives, responding to rendezvous protocol ready-to-send messages, executing user-level completion handlers, and refilling hardware receive queues.

By using a separate thread for these duties, we ensure that active message completion callbacks do not block progress on independent data transfers. While dedicating a CPU core to this job means that we lose computing capability, modern platforms can have hundreds of cores so any such loss is minor—on our experimental platform it is less than 1%—while the benefit of reduced communication latencies for both active messages and bulk data transport allows for a reduction in wasted idle time on worker threads and thereby improves overall performance.

5.3.2 Active Messages. The LCI backend for PaRSEC maintains a hash table that maps active message tags to callback handles. The callback handles contain the information necessary to execute an active message callback, such as the callback function pointer and the local callback data. Registering an active message tag simply inserts the relevant entry into the table.

Sending an active message is done using the LCI Immediate or Buffered communication calls, depending on the length of the message, so that all active messages are sent eagerly. This places an upper limit on the amount of data an active message can carry of about 12 KiB in the current implementation, but this is sufficient for the active messages utilized in PaRSEC. Buffers for receiving the active message are allocated dynamically by the LCI runtime from a PaRSEC memory pool at the destination. The destination does not need to post a corresponding receive or perform any sort of message matching.

After a message arrives at its destination, LCI invokes a handler function that looks up the active message handle from the message tag. A callback handle is allocated from a memory pool and filled with information specific to the active message, such as the buffer, rank of the source process, and callback function. The callback handle is then pushed to a shared FIFO queue to be consumed by the communication thread.

5.3.3 Data Transport. Bulk data transport is currently implemented as with the MPI backend, emulating a one-sided `put` using two-sided communications. The origin process sends a handshake message to the target telling it where data is to be received and what tag to use. Unlike the MPI backend, the active message infrastructure isn't used directly, but rather a specialized version of it is used; this lets us bypass operations such as the hash table tag lookup that would otherwise be required. This handshake contains the target address, the size of message data, the remote completion callback, the size of callback data, and the callback data itself. The tag of the handshake message encodes what tag is to be used for the communication. Depending on the size of the handshake message, it is sent using the LCI Immediate or Buffered communication calls.

The message data is then sent using the LCI Direct communication call. As in the MPI backend, since each tuple of $\langle Origin, Tag \rangle$ will be unique, MPI-style inter-message ordering is not required. A callback handle is allocated at the origin and filled with information regarding the ongoing communication, the local completion callback function, and its arguments. When the send completes, LCI will invoke a handler function that will push this callback handle to a shared FIFO queue to be consumed by the communication thread.

As an additional optimization, if the message data is sufficiently small, then it can be sent eagerly inside the handshake message. If the data was sent eagerly thus, a separate data communication is unnecessary and the local completion callback at the origin is called immediately.

At the target, when the handshake message arrives a handler function is invoked. The handler allocates a callback handle and fills it. It then attempts to start the Direct receive to match with the send at the origin. Since LCI allows this to fail with a "retry" error for any number of reasons, such as insufficient hardware resources, this must be handled appropriately. Since this is executed on the progress thread, we cannot simply retry until sufficient resources have been freed; nor can we invoke the LCI progress function to ensure progress without encountering severe issues with recursion. The solution is to delegate starting the matching receive to the communication thread in cases where the receive could not be immediately started by the progress thread. In most cases, except when the communication system is under significant pressure, this does not occur, allowing us to reduce the latency and match messages sooner. Once the posted receive completes, a

Table 1: SDSC Expanse hardware and software configuration.

Hardware		Software	
CPU	2× AMD EPYC 7742	OS	Rocky Linux 8.5
Cores	128 @ 2.25 GHz	Kernel	Linux 4.18.0-348.23.1
Memory	256 GiB DDR4	Compiler	AOCC 3.0.0
Storage	1TB local PCIe; Lustre	rdma-core	50mlnx1-1.49417
NIC	Mellanox ConnectX-6	MPI	Open MPI 4.1.5
Network	2× HDR InfiniBand	Backend	UCX 1.14.0
Topology	Hybrid Fat-Tree	LAPACK	Intel MKL 2019.5.281

completion handler is called in the LCI progress function. Similarly to the handler at the origin, the callback handle is pushed to the shared FIFO queue.

5.3.4 Completion Callback Progress. Progress on existing communications is handled solely by the progress thread. This leaves the communication thread the job of starting and completing communications. So as to ensure a level of fairness between processing completion callbacks for active messages and bulk data, we use separate FIFO queues for these types of communication. We remove up to five completion handlers from the active message queue, processing each of them in turn, followed by all available completion handlers in the bulk data queue. If any completions were processed this way, we loop, until no communications could be completed, after which we return control to the higher-level PaRSEC runtime.

6 PERFORMANCE RESULTS AND ANALYSIS

6.1 Experimental Setup

6.1.1 System Configuration. Performance results were gathered on the SDSC Expanse cluster. Each node has two sockets of AMD EPYC 7742 CPUs, each with 64 cores, for a total of 128 cores per node with 256 GiB memory. Nodes are connected to switches with 2 links of 50 Gbps HDR InfiniBand using a hybrid fat-tree topology. Additional system configuration details are shown in Table 1.

6.1.2 PaRSEC Configuration. For all tests with multiple nodes, we configure PaRSEC so that one core is dedicated to the communication thread and, for the LCI backend, another to the progress thread, as described in Sections 4.3 and 5.3. These are pinned to cores in the NUMA domain where the Mellanox ConnectX-6 NIC is attached. This is done to reduce communication latency: tests with free-floating communication and progress threads showed up to a 25% increase in mean end-to-end latency for communications as compared to using dedicated cores. All remaining cores are used for worker threads, so that MPI-based executions use 127 worker threads and LCI-based ones use 126. In single-node runs we use all the 128 cores for computation. Unless otherwise stated, we funnel Activate messages through the communication thread in addition to data transport.

For MPI support, we use Open MPI [16] with the `ucx` PML module. The default UCX parameters are used, with the exception of explicitly selecting the network device to use and setting `UCX_UNIFIED_MODE`, which increased and stabilized performance in some tests. We also set `UCX_IB_RCACHE_MAX_REGIONS` to a fixed limit; we found that using the default setting of uncapped cached registrations could breach the number of registrations allowed by the hardware under some scenarios, leading to a crash.

We have made available the versions of LCI[1] and PaRSEC[2] used in this paper. In Section 6.4 we demonstrate results with HiCMA[4], which depends on DPLASMA[3] [3].

6.1.3 Methodology. Measurements in Sections 6.2 and 6.3 were measured by running 18 executions in succession, discarding the first three, and computing the mean of the remaining 15 executions; we had observed that there was a significant difference between performance in the first three runs and the subsequent ones. Measurements in Section 6.4 were measured from a mean of five executions in succession.

In several experiments, we measure inter-node communication latency. To accurately obtain these measurements, we synchronize clocks with an algorithm adapted from [18]. We re-synchronize clocks at the beginning of every PaRSEC context execution epoch so as to prevent clock drift across several runs.

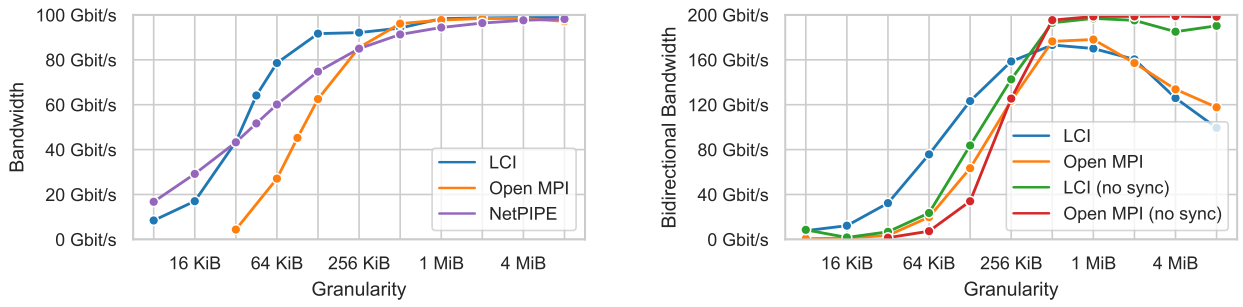
6.2 Ping-Pong Bandwidth

We utilize a task-based windowed ping-pong bandwidth microbenchmark in PaRSEC to compare the performance of the MPI backend and the new LCI backend. Each PingPong(t, f, c) task operates on a fragment f of size N. t defines the current iteration of the benchmark and the number of fragments defines the window size. Between each iteration, a synchronization task Sync(t) executes to force serialization. PingPong tasks execute round-robin between nodes, such that data communications occur—when running on two nodes, this results in the data traveling back and forth between them on the network. A number of independent "streams" can be started, beginning execution round-robin across the nodes, such that if P streams are started, where P is the number of nodes, then all nodes must send and receive data concurrently for each iteration t. c defines the stream of a PingPong task.

To demonstrate the efficacy of LCI in allowing PaRSEC to scale to smaller task sizes, we vary the size of each fragment between 8 MiB at 8 KiB while increasing the window size from 32 to 32,768 so as to keep the total amount of data in each iteration constant at 256 MiB. Since the PaRSEC runtime core is unchanged, the task management overhead must be identical, so differences in performance must be due to communication management.

Performance results with one stream are shown in Figure 2a. We compare PaRSEC bandwidth results to NetPIPE [29] as a baseline for ping-pong bandwidth on an InfiniBand cluster. We see that when tasks are coarse-grained, both PaRSEC backends are able to achieve peak bandwidth, but that as tasks become smaller and greater in number, performance decreases. However, LCI maintains near-peak performance at smaller granularity than the MPI backend and is able to support smaller task sizes more efficiently: while the MPI backend drops to 62.5 Gbit/s at 128 KiB and to 45.2 Gbit/s at 90.5 KiB the performance of the LCI backend is reduced to 64.1 Gbit/s only at a granularity of 45.25 KiB and to 43.5 Gbit/s at 32 KiB, supporting tasks about 2.83 times smaller at a similar efficiency.

[1] https://github.com/uiuc-hpc/LC/releases/tag/icpp23
[2] https://github.com/uiuc-hpc/parsec/releases/tag/icpp23
[3] https://github.com/uiuc-hpc/dplasma/releases/tag/icpp23
[4] https://github.com/uiuc-hpc/hicma-parsec/releases/tag/icpp23



(a) One Stream **(b) Two Streams**

Figure 2: PaRSEC bandwidth benchmarks. With two streams, performance with large tasks is decreased due to a queuing effect that is eliminated by removal of inter-iteration synchronization.

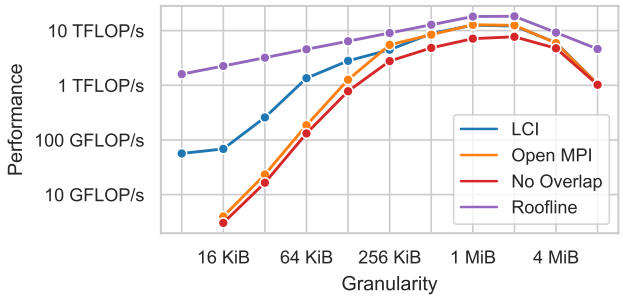


Figure 3: Overlap benchmark with GEMM-like intensity.

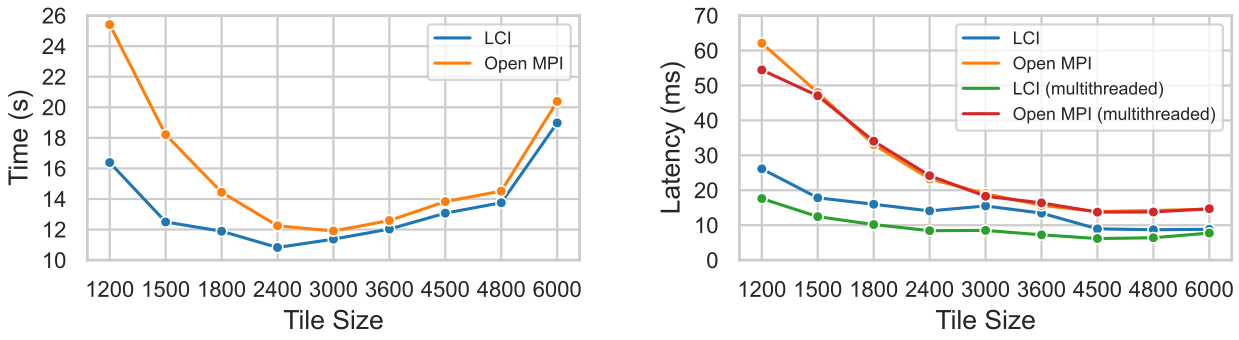
Bidirectional bandwidth results are in Figure 2b. We see that similarly to the one-stream case, the LCI backend can scale more efficiently to a smaller task granularity. Performance results with large message sizes are somewhat surprising, as we would expect that it is easier to achieve peak bandwidth with larger messages. We attribute this reduction in performance to a queuing phenomenon where, due to the aforementioned synchronization and the small number of fragments, one stream can overtake the other and both streams end up traveling together, so that each node is either only sending or receiving at any given point in time. By loosening the synchronization requirement so that the data for the next iteration can be sent while the prior iteration is still executing, we find that we recover the "lost" performance, with both backends able to achieve near-peak bidirectional bandwidth. However, the reduction in inter-iteration synchronization results in an increase in the number of Activate messages, as fewer are aggregated as described in Section 4.3; this increase is particularly notable for fine-grained tasks, so that less network bandwidth is available for data movement as compared to the tightly-synchronized benchmark.

6.3 Computation/Communication Overlap

We next consider how efficiently PaRSEC is able to overlap computations for independent tasks and communications for future tasks. For this purpose, we use a variant of the bandwidth benchmark described in Section 6.2 with the ability to execute a configurable number of double-precision floating-point multiply-accumulate operations on each 8-byte portion of the fragment belonging to a Ping-Pong task. By allowing the intensity to be configurable, we can independently vary the time required to execute each task and the amount of data each task operates on and that therefore must be communicated per iteration.

For the overlap test, we consider tasks with a compute intensity similar to GEMM. A GEMM executes N^3 FMA operations for N^2 elements, so there are N operations per element. To achieve a similar intensity, a Ping-Pong task operating on M bytes must execute $\sqrt{M/8}$ FMA operations per 8 bytes. To ensure that the same total number of FLOPs are executed regardless of task granularity and number of fragments, we increase the number of iterations to compensate for the decreased intensity of each task. This means that while the amount of data and the number of FLOPs are constant, the amount of data *moved* across the network increases as the task granularity decreases. This trade-off is comparable to those in many real applications: smaller task size decreases the computation-to-communication ratio. To increase the likelihood of overlap, we remove the Sync task described in the prior section.

Figure 3 shows the performance results for overlapping tasks with GEMM-like intensity with communications. The "Roofline" curve simulates performance assuming that communication is entirely overlapped with computation, while "No Overlap" simulates performance assuming that no communication can execute concurrently with computation. When tasks are large, performance is limited by the number of tasks that can be executed in parallel. Since we retain the 256 MiB total data used in the bandwidth benchmarks, when fragments are 8 MiB in size there are only 32 tasks that can be executed in each iteration per stream. As tasks decrease in size, but increase in number, performance begins to be limited by the computing performance of the system. Then as task size decreases further, the network bandwidth begins bounding performance. Once the task size has grown sufficiently small, the MPI backend begins to struggle to move the data fast enough, while the LCI backend continues to keep pace with the shrinking data size. At the 128 KiB fragment size, the LCI backend is able to achieve over twice the performance of the MPI backend, while at 32 KiB it is an order of magnitude faster.



(a) Time-to-solution.

(b) End-to-end communication latency.

Figure 4: TLR Cholesky, N = 360,000, 16 nodes, scaling tile size from 6000 × 6000 to 1200 × 1200. Latency is measured from send of the Activate message to arrival of data for individual flows. "MT" indicates that communication multi-threading for Activate messages is enabled.

6.4 HiCMA

6.4.1 Background. Sparse linear algebra is an important HPC application domain where application performance is often sensitive to communication. We experiment with HiCMA, a tile-based low-rank Cholesky approximation code that is used in applications such as geostatistical modeling [6]. HiCMA first compresses off-band tiles of a large matrix to a low-rank representation while maintaining a specified accuracy threshold, then uses PaRSEC to orchestrate the execution of a task-based Cholesky factorization where kernels can operate directly on the low-rank tile format. Additional details on the operation of HiCMA are available at [7, 8].

HiCMA poses several unique challenges to a dynamic runtime. The compute kernels are regular in their data access pattern as is the overall structure of the task graph, but the low-rank GEMM kernels that make up the bulk of the tasks are far less compute-intense than traditional GEMM kernels. When there are sufficiently many tasks—which is to say, that tiles are sufficiently small—there should always be enough parallelism to ensure that cores are not idle. However, decreasing the tile size also somewhat decreases the resulting tasks' already-low compute intensity, so that if tasks are *too* small, moving data fast enough to ensure that ready tasks are always available becomes a significant bottleneck to performance.

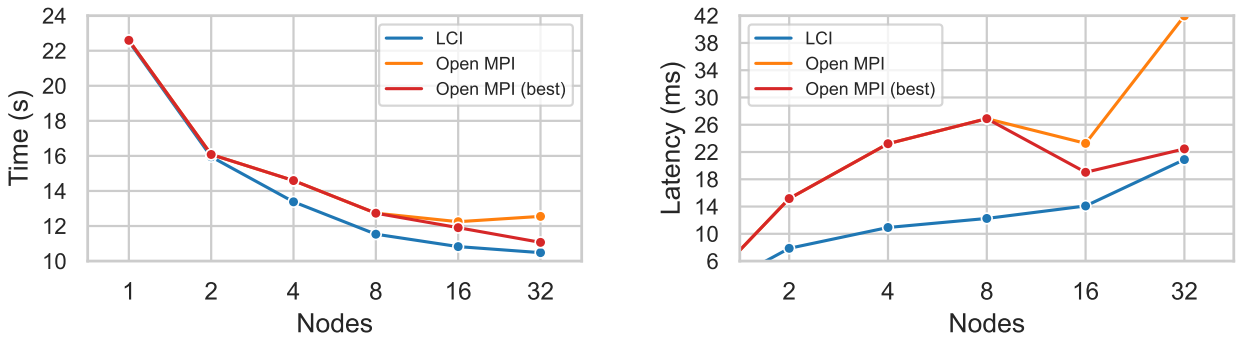
Additionally, message sizes can vary widely. Dense tiles on the diagonal band are very large and can easily saturate network bandwidth alone, while low-rank tiles far from the diagonal can see their rank drop to 1, so that they may be only a few tens of kilobytes in size. With each tile having a different rank, and the inherent dynamism of the PaRSEC runtime, significant pressure is asserted against the communication backend to ensure data is delivered in a timely manner. The key element to achieving the best performance is guaranteeing that data for the "critical path" tasks that are direct predecessors of the panel operations on the dense diagonal tiles arrives without delay, so that maximum parallelism is maintained.

6.4.2 Tile Scaling. We first consider how changing task granularity affects performance while keeping the number of nodes constant. As described above, there are several factors working against achieving the best possible performance at all tile sizes. If tiles are too large, there may be too few tasks to saturate all cores. For instance, for matrix size 360,000 × 360,000, at a tile size of 6000 × 6000, there are 60 tiles per dimension, for 1770 tiles in total, on which operate 37,820 tasks. However, only a fraction of them can execute in parallel—about 630 or so. If we were to scale with this tile size to even a few nodes, we would have many cores idle due to a lack of parallelism. However, at the other extreme, while there is sufficient task parallelism, the runtime may struggle to move the data fast enough to ensure that tasks can begin execution. It is clear that the best performance will be obtained at some midpoint between these two and achieving best performance at a smaller tile size is an indication of better scalability.

Figure 4 demonstrates this principle. We run the HiCMA st-2d-sqexp problem type on a matrix of size 360,000 × 360,000 at varying tile sizes between 6000×6000 and 1200×1200 using *maxrank* = 150 and a fixed accuracy of 10^{-8} with a band size of 1 and using the two-flow HiCMA algorithm [7, 8]. We quickly see in Figure 4a that, as anticipated, using a large tile size has reduced performance due to insufficient parallelism. We also observe that LCI is able to obtain a lower time-to-solution at all tile sizes, demonstrating the efficacy of reducing communication latency. As expected, this improvement diminishes at larger tile sizes, as communications are larger and their latency is predominantly bound by the network hardware, so reducing software overheads has less impact on time-to-solution.

Figure 4b shows the average end-to-end latency, as measured from when an Activate message is sent following task completion until the data arrives at the destination, taking into account the entire multicast tree. LCI achieves a lower mean end-to-end latency at every tile size, tracking closely with the behavior of the overall time-to-solution, particularly at the very small tile sizes where the communication backend must handle sending very many small tiles: the average rank is 10.44, so tiles in packed $U \times V$ format consume about 196 KiB of memory on average—and the largest low-rank tile is only 544 KiB, with a rank of 29.

6.4.3 Communication Multithreading. Figure 4b also shows the effect on mean communication latency when enabling multithreaded



(a) Time-to-solution.

(b) End-to-end communication latency.

Figure 5: TLR Cholesky, N = 360,000, scaling number of nodes from 1 to 32. Tile size is decreased while scaling to ensure sufficient parallelism. "Open MPI (best)" indicates where achieving best performance with Open MPI required coarsening tasks by increasing tile size. Latency is measured from send of the ACTIVATE message to arrival of data for individual flows.

Table 2: Tile Size with Lowest Time-to-Solution.

Nodes	1	2	4	8	16	32
Open MPI	4500	4500	3600	3000	3000	3000
LCI	4500	4500	3600	3000	2400	1800

communication support in PaRSEC. This support allows computing threads to send ACTIVATE messages themselves rather than delegating to the communication thread. While using this multithreading support disables aggregation of such messages, it also significantly reduces the latency of communications when using the LCI backend: the mean latency of individual messages in the multicast is reduced by up to 63%, and end-to-end latency is reduced by up to 46%. This translates into a 10% speedup in time-to-solution for the 1200 tile size, from 16.384 to 14.839 seconds. For the best-performing 2400 tile size, communication multithreading results in a 3% improvement in overall performance, reducing the running time to 10.516 seconds.

When using the MPI backend, communication performance is generally neutral or negatively impacted. The most significant improvement is at the 1200 tile size, which saw a 18% reduction in latency from the direct multicast predecessor and a 12% reduction in end-to-end latency from the multicast root. Other tile sizes saw no significant change.

6.4.4 Strong Scaling. We use the same $N = 360{,}000$ problem size and perform a strong scaling test, keeping the total work constant while increasing the number of nodes. Results are reported in Figure 5. Tile sizes where LCI and Open MPI achieved the best time-to-solution are reported in Table 2. When using fewer nodes, using larger tiles is optimal; but as the number of compute cores increases, the number of tasks must also increase to maintain sufficient parallelism, necessitating a decrease in the tile size. In Figure 5a we report the time-to-solution for two sets of tile sizes when using Open MPI: the same tile size where LCI achieved the best performance and the best-performing tile size for Open MPI. Because the LCI backend is able to obtain better bandwidth and latency for smaller data, we are able to scale to smaller tile sizes and a greater node count.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented evidence that widely-used communication substrates such as MPI are ill-suited to achieving the best performance in dynamic runtimes. We present an improved design for a Lightweight Communication Interface that more closely matches the needs of asynchronous tasking runtimes and demonstrate how we integrate it with the new PaRSEC communication API. To determine the efficacy of our approach, we highlighted improvements in several areas. First, we demonstrated improved small-task bandwidth in PaRSEC, allowing the use of tasks nearly three times smaller with similar efficiency. Next, we showed how we can achieve better overlap between computations and communications, resulting in performance over one order of magnitude faster than with the MPI backend. Finally, we extended these approaches to a state-of-the-art TLR Cholesky factorization, where we demonstrated that using the LCI backend was able to reduce mean end-to-end latency for communications by over 50%, resulting in a net speedup in time-to-solution of 12%. For future work, we plan on introducing new features to LCI that can directly implement the PaRSEC `put` interface and examining the benefits of using multiple communication or progress threads to further reduce communication latency in highly-loaded scenarios. We also would like to examine opportunities for closer integration between dynamic runtimes and LCI, leveraging abilities to quickly enable tasks or lightweight threads, or executing high-priority short-duration tasks in the context of the progress thread. This would enable new classes of applications that are currently bottlenecked by task runtime overheads.

ACKNOWLEDGMENTS

This research was supported by NSF grants 1908144 and 1909015. This work used the Expanse system at the San Diego Supercomputer Center through ACCESS allocation CCR130058.

REFERENCES

[1] 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

[2] 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

[3] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *25th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2011)*. IEEE, 1432–1441. https://doi.org/10.1109/IPDPS.2011.299

[4] 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

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Comput.* 38, 1 (Jan. 2012), 37–51. https://doi.org/10.1016/j.parco.2011.10.003

[6] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. 2022. Reshaping Geostatistical Modeling and Prediction for Extreme-Scale Environmental Applications. In *2022 International Conference for High Performance Computing, Networking, Storage and Analysis (SC22)*. IEEE, 1–12. https://doi.org/10.1109/SC41404.2022.00007

[7] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2021. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2021)*. IEEE, 79–89. https://doi.org/10.1109/IPDPS49936.2021.00017

[8] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2020. Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Platform for Advanced Scientific Computing Conference (PASC '20)*. ACM, 1–11. https://doi.org/10.1145/3394277.3401846

[9] Franck Cappello, Amina Guermouche, and Marc Snir. 2010. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (ICCCN 2010)*. IEEE, 1–8. https://doi.org/10.1109/ICCCN.2010.5560143

[10] Jaemin Choi, Zane Fink, Sam White, Nitin Bhat, David F. Richards, and Laxmikant V. Kalé. 2021. GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python. In *35th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2021)*. IEEE, 479–488. https://doi.org/10.1109/IPDPSW52791.2021.00079

[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] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards millions of communicating threads. In *23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, 1–14. https://doi.org/10.1145/2966884.2966914

[13] Hoang-Vu Dang, Marc Snir, and William Gropp. 2017. Eliminating contention bottlenecks in multithreaded MPI. *Parallel Comput.* 69 (Nov. 2017), 1–23. https://doi.org/10.1016/j.parco.2017.08.003

[14] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 752–768. https://doi.org/10.1145/3192366.3192404

[15] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *28th International Conference on Parallel Architectures and Compilation Techniques (PACT 2019)*. IEEE, 15–28. https://doi.org/10.1109/PACT.2019.00010

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting*. Springer, 97–104. https://doi.org/10.1007/978-3-540-30218-6_19

[17] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. 2015. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 34–39. https://doi.org/10.1109/HOTI.2015.19

[18] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Hierarchical Clock Synchronization in MPI. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 325–336. https://doi.org/10.1109/CLUSTER.2018.00050

[19] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. 2020. HPX - The C++ Standard Library for Parallelism and Concurrency. *Journal of Open Source Software* 5, 53 (Sept. 2020), 2352. https://doi.org/10.21105/joss.02352

[20] Laxmikant V. Kalé and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. ACM, 91–108. https://doi.org/10.1145/165854.165874

[21] Sameer Kumar, Yanhua Sun, and Laximant V. Kalé. 2013. Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q. In *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*. IEEE, 689–699. https://doi.org/10.1109/IPDPS.2013.83

[22] 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

[23] MPI Forum. 2021. MPI: A Message-Passing Interface Standard Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

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

[25] Joseph Schuchart, Christoph Niethammer, José Gracia, and George Bosilca. 2021. Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication. https://doi.org/10.48550/arXiv.2111.08142

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

[27] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

[28] 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*. IEEE, 40–43. https://doi.org/10.1109/HOTI.2015.13

[29] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. 1995. NetPIPE: A Network Protocol Independent Performance Evaluator. In *1995 IASTED International Conference on Intelligent Information Management and Systems (IIMS 1995)*. IASTED/ACTA, 1–6.

[30] 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

[31] Yanhua Sun, Gengbin Zheng, Laximant V. Kalé, Terry R. Jones, and Ryan Olson. 2012. A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect. In *26th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012)*. IEEE, 751–762. https://doi.org/10.1109/IPDPS.2012.127

[32] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. IEEE, 1–8. https://doi.org/10.1109/IPDPS.2008.4536359