Examining MPI and its Extensions for Asynchronous Multithreaded Communication

Jiakun Yan[1][0000−0002−6917−5525], Marc Snir[1][0000−0002−3504−2468], and Yanfei Guo[2][0000−0002−3731−5423]

[1] University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
{jiakuny3,snir}@illinois.edu
[2] Argonne National Laboratory, Lemont, IL 60439, USA
yguo@anl.gov

Abstract. The increasing complexity of HPC architectures and the growing adoption of irregular scientific algorithms demand efficient support for asynchronous, multithreaded communication. This is most pronounced with Asynchronous Many-Task (AMT) systems. Such communication was not a consideration during the initial MPI design. The MPI community has recently introduced several extensions to address these new requirements. This work evaluates two such extensions, the Virtual Communication Interface (VCI) and the Continuation extensions, in the context of an established AMT runtime, HPX. We begin by using an MPI-level microbenchmark, modeled from HPX's low-level communication mechanism, to measure the peak performance potential of these extensions. We then integrate them into HPX to evaluate their effectiveness in real-world scenarios. Our results show that while these extensions can enhance performance compared to standard MPI, areas for improvement remain. The current continuation proposal limits the maximum multithreaded message rate achievable in the multi-VCI setting. Furthermore, the recommended one-VCI-per-thread mode proves ineffective in real-world scenarios due to the attentiveness problem. These findings underscore the importance of improving intra-VCI threading efficiency to achieve scalable multithreaded communication and fully realize the benefits of recent MPI extensions.

Keywords: Multithreaded communication · Asynchronous communication · Task parallelism · VCI · continuation.

1 Introduction

High-performance computing (HPC) architectures are becoming increasingly heterogeneous with extensive on-node parallelism and deep memory hierarchies. Modern compute nodes often feature over 100 CPU cores and multiple accelerators. Meanwhile, scientific applications are adopting more adaptive or sparse algorithms [20, 26] to achieve higher resolution and scalability. These trends challenge the traditional Bulk-Synchronous Parallel (BSP) model, in which all processes operate in lockstep with evenly distributed workloads.

Asynchronous Many-Task (AMT) systems have emerged as a compelling alternative. In these systems, applications are expressed as task dependency graphs, and the runtime manages task scheduling, dependencies, and communication, usually operating with one multithreaded process per socket or node. AMT runtimes employ oversubscription, asynchronous execution, and communication-computation overlap to outperform hand-tuned BSP implementations in irregular workloads [1, 40, 14].

AMTs exhibit different communication characteristics from BSP applications [28, 42]. Messages are typically finer-grained and dominated by point-to-point communication rather than global collectives. Communication patterns are highly dynamic, with many outstanding operations, and most threads (logically or physically) can generate or consume messages. These characteristics fall outside the traditional design and optimization focus of MPI.

This paper investigates how well existing MPI and recent extensions can support AMT's communication requirements through a case study of an established AMT runtime, HPX. While our focus is on AMTs, their communication challenges are increasingly common in applications with data-dependent execution, beyond the traditional BSP domain. To remain broadly relevant, MPI must evolve to meet these demands.

Building on the analysis of communication requirements of AMT presented in [42], we focus on two critical features shown to impact application-level performance significantly: (1) scalable handling of many concurrent communication operations, and (2) effective replication of communication resources to reduce contention. We first use an MPI-level microbenchmark, modeled from HPX's low-level communication mechanism, to evaluate the raw capabilities and limitations of the tested extensions, and then integrate them into HPX to assess their practicality and system-level effectiveness.

Specifically, our evaluation is based on MPICH [31] and primarily involves two MPI extensions:

- *The Virtual Communication Interface (VCI) Extension* [44]: a mechanism to mitigate thread contention by replicating internal communication resources and mapping them to distinct communicators.
- *The Continuations Extension* [37]: a callback-based completion mechanism designed to reduce the overhead of managing large numbers of pending operations.

Our results reveal both these extensions' advantages and current limitations and motivate recommendations for evolving MPI standards and implementations to better support asynchronous multithreaded runtimes.

The rest of the paper is organized as follows. Section 2 provides background on the MPI threading model and the extensions we study. Section 3 describes how we integrate the VCI and continuation extensions into the HPX parcelport logic, including our modifications to the existing extensions. Section 4 presents our MPI-level microbenchmark and the fundamental performance characteristics of the extensions. Section 5 then evaluates the extensions in the context of the HPX runtime, using both microbenchmarks and a real-world astrophysics application,

OctoTiger [14]. Section 6 presents related work. Finally, Section 7 concludes the paper and discusses suggestions for improving MPI support for AMT systems.

2 Background

2.1 MPI Threading Level

The MPI specification [30] defines four levels of thread support, in increasing order: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. `MPI_THREAD_MULTIPLE` offers the highest level of thread support, allowing multiple threads to invoke MPI functions simultaneously. This model is the most intuitive approach for writing multithreaded MPI programs and is preferred by many users [7]. Most AMT systems such as HPX [23], Legion [6], and Charm++ [24] rely on `MPI_THREAD_MULTIPLE`. However, efficient support for this thread level has historically been lacking in many MPI implementations [35], primarily due to contention on internal MPI data structures and underlying network resources. This work focuses on optimizing and evaluating MPI extensions, specifically in the context of `MPI_THREAD_MULTIPLE`.

2.2 MPICH VCI

The Virtual Communication Interface (VCI) extension [44] is a mature mechanism in MPICH for addressing the MPI multithreaded efficiency issue through resource replication. When enabled, the MPICH runtime will associate a distinct set of communication resources (VCIs) with every MPI communicator, allowing each thread to communicate on dedicated resources with minimal contention. It also has advanced options for mapping communications to different VCI using message tags. MPICH recommends that multithreaded applications allocate a separate VCI/communicator for each thread. However, as we will demonstrate in this paper, this setup does not always yield optimal performance in real-world scenarios.

The design and implementation of VCI have been covered in detail in [44, 46]. As a brief overview, a VCI represents a relatively independent set of communication resources needed on the critical path of MPI communication routines. It primarily includes a UCP worker (when using the UCX [38] backend) or an OFI domain (when using the OFI [33] backend), which further encapsulates resources related to network hardware interfacing, memory registration, tag matching, and progressing. MPICH employs a per-VCI spinlock to ensure thread safety, allowing concurrent operations across VCIs but serializing accesses within each VCI.

2.3 MPI Continuations Proposal

The MPI Continuations Proposal [36] aims to provide an efficient mechanism for managing multiple pending communication operations. In standard MPI,

the only way to track pending operations is to wait for or test the request object corresponding to each communication operation. However, in event-driven systems such as AMTs, threads may post many communication operations concurrently, and the runtime must react when any individual operation completes. `MPI_Testsome` is unsuitable for this use case as it is typically implemented as a loop over the input request array, and maintaining the request array is inconvenient and expensive. Instead, such systems typically maintain lists of MPI requests (i.e., request pools) and use `MPI_Test` to opportunistically probe requests in the pool until one or more completed ones are found. A thread polls the pool when it becomes idle [13, 27, 41].

To avoid the polling overhead and the thread synchronization required to manage shared request pools, the continuation proposal introduces an API that allows MPI clients to attach callback functions to individual requests and register them with a *continuation request*. The application then polls only this continuation request to drive progress, and callbacks are invoked automatically when corresponding communication operations complete. [37] implements this proposal on a test branch of OpenMPI and integrates it into PaRSEC, operating in a mode where a single communication thread handles all MPI calls.

In this work, we implement the continuation proposal in MPICH and evaluate it in the context of HPX, where all worker threads can produce and consume messages concurrently. This context provides a more realistic test case for multithreaded communication. We further investigate how well the continuation mechanism integrates with multi-VCI configurations and assess its effectiveness in managing completion overhead in these scenarios.

3 Extend HPX parcelport with MPI Extensions

In this section, we will describe the design of HPX's low-level communication layer, known as *parcelport*, and detail how we integrate the two MPI extensions into the parcelport implementation. We also discuss the modifications made to the continuation extension to better support multi-VCI scenarios.

3.1 Background

We first briefly describe the HPX communication stack and the original MPI parcelport implementation. Please refer to [42] for a more detailed description.

HPX Application Interface HPX provides a rich set of APIs for developing parallel and distributed applications. At the core of its distributed programming model is a Remote Procedure Call (RPC) mechanism. Users can register functions or class methods as `Actions`, and allocate globally accessible objects. Any HPX process can then invoke these actions remotely, either on another process or on a global object.

HPX Communication Stack Overview Currently, HPX has three fully functioning communication backends: TCP, MPI, and LCI [41]. HPX's communication stack is organized into two layers. The *upper layer* is shared by all backends and handles essential services such as action argument (de)serialization, global object address resolution, message aggregation, and termination detection. Below it, the *parcelport layer* is backend-specific and implements the actual data transfer protocol.

In HPX, messages are transmitted in the form of *parcels*, each representing one or multiple remote action invocations and (logically) consisting of one non-zero-copy (NZC) chunk and an optional set of zero-copy (ZC) chunks. The NZC chunk contains control metadata, while the ZC chunks hold bulk data. This design avoids expensive memory copying by separating metadata from large payloads. Since ZC chunks must be deserialized into memory layouts compatible with C++ data structures, the upper layer pre-allocates appropriate receive buffers before the parcel is fully received.

Each parcelport must implement two core functions: (1) a non-blocking `send_parcel` function to send parcels and invoke a callback when complete, and (2) a `background_work` function that checks for incoming parcels and progresses outstanding communication. The `background_work` function is frequently invoked by idle threads and notifies the scheduler whether communication made forward progress. It passes the received parcels to the upper layer by enqueuing or immediately executing the encapsulated tasks.

Baseline MPI Implementation The original MPI parcelport transfers an HPX parcel using a sequence of MPI messages, consisting of a header followed by one or more data messages. The header contains metadata, such as NZC size, number of ZC chunks, and the MPI tag used for the follow-ups, and may piggyback the NZC chunk if it is small enough. Each remaining chunk is sent in a separate message.

All communications are non-blocking. Header and data messages use `MPI_Isend`, with a common tag for header messages and a distinct tag for each parcel. A single `MPI_Irecv` (pre-posted with `MPI_ANY_SOURCE` and the header tag) listens for incoming headers. Upon receiving one, the receiver posts additional `MPI_Irecv`'s for the corresponding data messages, using buffer allocations from the upper layer as needed.

To simplify synchronization, each parcel has at most one active `MPI_Isend` or `MPI_Irecv` at a time; the following message is posted only after the current one completes. Messages from different parcels may proceed concurrently. The MPI request handles for pending sends and receives (except the preposted receive) are stored in two STL deques (*request pools*). The `background_work` function is responsible for polling the preposted receive request and the request pools using `MPI_Test`. The request pools are polled in a round-robin fashion.

Because any HPX worker may call `send_parcel` and `background_work`, the parcelport must be thread-safe. MPI is initialized with `MPI_THREAD_MULTIPLE`, and all polling operations are guarded by an HPX lock. The parcelport will

use non-blocking `try-lock` whenever possible. In the case of blocking waiting, the HPX lock will deschedule the underlying user-level threads to avoid wasting CPU cycles.

3.2 Replication of Communicators

The baseline implementation uses a single communicator. In MPICH, this maps to a single set of internal communication resources and is protected by a single spinlock. This causes severe thread contention for the lock if multiple threads access it simultaneously to post sends/receives or test corresponding requests. The VCI extension in MPICH enables us to replicate internal communication resources by mapping them to distinct communicators. We thus enhance the baseline MPI parcelport with the ability to split communication traffic into a configurable number of communicators. We will call this enhanced parcelport the *MPIx parcelport*.

We must ensure the send and receive operations for the same MPI message are posted with the same communicator. Therefore, we construct a static mapping from HPX worker threads to MPI communicators during parcelport initialization. We assign HPX worker threads to communicators in an order that ensures most adjacent threads are assigned the same communicator, thereby improving locality. When the upper layer invokes the `send_parcel` function of the parcelport layer on a worker thread, the following MPI send and receive calls for that parcel will use the communicator associated with this thread. The header message carries the index of this communicator.

With multiple VCIs, the MPIx parcelport will pre-post one `MPI_Irecv` for incoming header messages for each communicator. Worker threads will poll for completed communications using `background_work` only for their communicator. When the continuation extension is not used, the request pools are replicated per communicator, and the `background_work` function will poll the request pools associated with their communicator.

The current MPICH implementation employs a hybrid progress model in the case of multiple VCIs: a progress call (happening implicitly inside `MPI_Test` and all blocking MPI functions) will primarily progress the VCI that is associated with the calling operation, but it will also progress all VCIs once in a while (every 255 VCI-local progress calls). This provides stronger progress guarantees [45], but also increases contention between threads. As a result, we set the `MPIR_CVAR_CH4_GLOBAL_PROGRESS` to `false` to turn off the occasional global progress. Section 4.3 analyzes the performance impact of this setting.

3.3 Replacing Request Polling with Callbacks

The Continuations Proposal allows clients to attach a callback function to an operation request. In the MPIx parcelport, after we post a `MPI_Isend` for a header or follow-up message or a `MPI_Irecv` for a follow-up message, we attach a callback function to the resulting request. The callback function will push a completion descriptor to a preallocated completion queue. Essentially, we use the

continuation callback to implement a queue-based completion mechanism. The `background_work` function will poll the completion queue for any completed operation and react accordingly. We share the completion queue among all threads to improve load balancing. The completion queue uses a state-of-the-art atomic queue implementation (LCRQ [29]).

We do not directly invoke the HPX completion logic in the callback because HPX can invoke arbitrary user tasks and even destroy the current user-level thread, which can lead to reduced performance and even deadlocks. The queue-based design decouples the upper-level complexity from the low-level communication logic.

One side effect of sharing the queue across threads is potential contention during follow-up `MPI_Isend` and `MPI_Irecv` operations, as these may be issued by threads not originally associated with the relevant communicator. However, such contention only occurs for large parcels, which are assumed to be relatively infrequent and less contention-sensitive. Prior experiments have shown that the benefits of using a shared queue typically outweigh this overhead.

Complication with Continuation Requests: While the core mechanism of the Continuations Proposal is to attach callback functions to individual MPI operation requests, it is not the entire proposal. To ensure progress and allow more controls over callback execution, the proposal also introduces a persistent *continuation request* object. All continuations (requests with attached callbacks) must be registered with a continuation request. The continuation request is marked complete when all the registered continuations have executed; the continuation request can be tested for completion, and has to be explicitly restarted with `MPI_Start` before newly attached continuations can be executed again. In MPICH, an atomic counter per continuation request tracks the total number of pending requests to determine whether the continuation request is complete.

The continuation proposal expects users to test the continuation requests to drive the MPI progress engine. In a multi-VCI setup, when a continuation request is tested, the MPI runtime must determine which VCI(s) to make progress on. MPICH adopts the following strategy for selecting the VCI(s) to make progress: Each continuation request maintains a per-VCI atomic counter to track the number of pending operations on that VCI; when testing the continuation request, the MPICH implementation will only make progress on VCI(s) with active associated operations (along with occasional global progress).

In many scenarios, the continuation request functionality adds unnecessary overhead: progress can be guaranteed using other MPI calls, and each communication completion already invokes a client-defined callback. From the client's perspective, there is no need to test for the completion of multiple handler invocations explicitly. Therefore, we extend the existing continuation proposal with the option to disable the usage of the continuation request, by setting the `cont_request` argument to `MPI_REQUEST_NULL` in the `MPIX_Continue` function. In this case, we can avoid the overhead of atomically counting the pending callbacks and completing/restarting the continuation request. We evaluate the performance implications of this optimization in Section 4.4.

In HPX, we adopt this optimization and skip the allocation of continuation requests entirely. HPX worker threads periodically poll their pre-posted receives, which automatically invokes the progress engine for the corresponding VCI. It is a lovely coincidence that HPX does not need to do anything additional to ensure the progress of all pending communications attached to continuation callbacks. For other clients where this is not the case, the MPICH runtime provides a non-standard function `MPIX_Stream_progress` to invoke the progress engine of a specific VCI explicitly.

4 MPI-level Microbenchmark

We begin with a multithreaded active message ping-pong microbenchmark to evaluate the basic performance characteristics of the mechanisms used in the MPIx parcelport, independent of the HPX runtime. To do so, we isolate the active message layer from the MPIx parcelport implementation in HPX and use it to construct a standalone microbenchmark. The active message layer uses pre-posted receives for incoming messages and manages the pending sends with request pools, with the option to leverage the VCI and continuation requests. The benchmark runs on two nodes, each hosting a single MPI process with a configurable number of threads. Threads are pinned to individual cores, and each thread performs a fixed number of ping-pong iterations with a corresponding peer thread on the remote node. All communications use the active message services provided by the extracted layer.

The isolated active message layer organizes the relevant MPI resources (including a communicator, a preposted receive request, and a request pool) into a logical unit called a *device*. All threads share a single device in the baseline (standard MPI) configuration. With the VCI extension enabled, each thread is assigned a private device, mapped to a distinct VCI. With the continuation extension, request pools are replaced with callbacks.

4.1 Experiment Setup

We run all the experiments in this section and Section 5 on SDSC Expanse and NCSA Delta. Table 1 summarizes the platforms' configurations. The two platforms have similar CPUs but have different network hardware and software stacks. Expanse uses HDR InfiniBand with Mellanox ConnectX-6 NICs, while Delta uses HPE Slingshot-11 with HPE Cassini NICs. On InfiniBand, MPICH can use either UCX [38] or OFI [33] as the network backend, while on Slingshot-11, MPICH can only use OFI. We use a customized version of MPICH 4.3.0 that implements the continuation proposal. This version is currently available in a pull request on the MPICH GitHub repository[3].

[3] https://github.com/pmodels/mpich/pull/7164

Table 1: Platform Configuration.

Platform	SDSC Expanse	NCSA Delta
CPU	AMD EPYC 7742	AMD EPYC 7763
sockets/node	2	2
cores/socket	64	64
NIC	Mellanox ConnectX-6	HPE Cassini
Network	HDR InfiniBand (2x50Gbps)	Slingshot-11 (200Gbps)
Software	MPICH 4.3.0 UCX 1.17.0 Libfabric 1.21.0 OpenMPI 4.1.3 Libibverbs 43.0	MPICH 4.3.0 Cray MPICH 8.1.27 Libfabric 1.15.2.0 SSHOT2.1.3

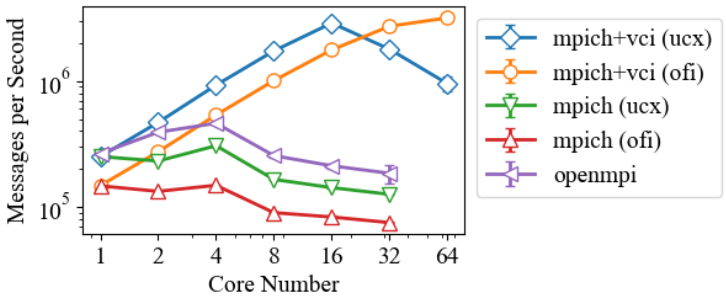
4.2 Overall Performance with Multiple VCIs

We begin by evaluating the performance impact of using multiple VCIs with different MPICH network backends and compare the results to those of system-installed MPI implementations (OpenMPI and Cray-MPICH) and standard MPICH without VCI extensions.

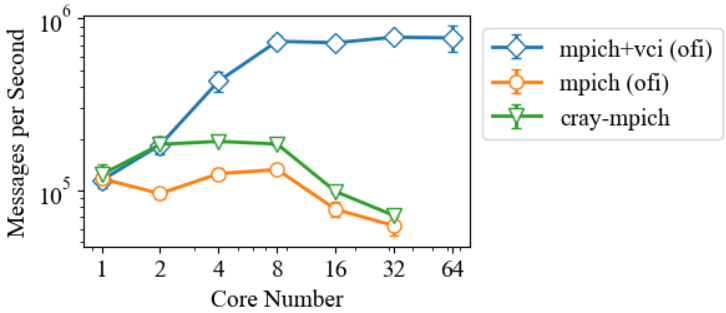
As shown in Fig. 1, the MPICH VCI extension improves the multithreaded performance of MPI, outperforming both the system-installed MPI (OpenMPI and Cray-MPICH) and standard MPICH itself. When comparing the best-performing multi-VCI configurations against the best standard MPI configurations using 32 threads per process, we observe speedups of 10x on Expanse and 8x on Delta. Reduced thread contention through replicated communication resources is the primary cause of the speedup. However, the performance gain depends on the underlying network backend, revealing a trade-off between UCX and OFI. While UCX has better base performance, it scales poorly when the number of threads/VCIs exceeds 16. On Expanse with 64 threads (and 64 VCIs), MPICH with the OFI backend outperforms the UCX backend by 4×.

In the standard MPI configuration shown in Fig. 1, all threads share a single device (i.e. a communicator, a preposted receive, and a request pool). For comparison, we also evaluated a variant where each thread has its own device, still using standard MPI. However, it results in even lower performance than the shared device case. With multiple outstanding pre-posted receive requests, there is more contention for the blocking lock of the VCI.

We have also compared the continuation extension's performance against plain request polling. However, we found no performance difference between the two approaches in either the multi-VCI or the standard MPI cases. This is expected, as in this ping-pong microbenchmark, each thread has only one send request and one receive request to poll simultaneously.



(a) Experiment Results on Expanse with 1-64 threads per process.

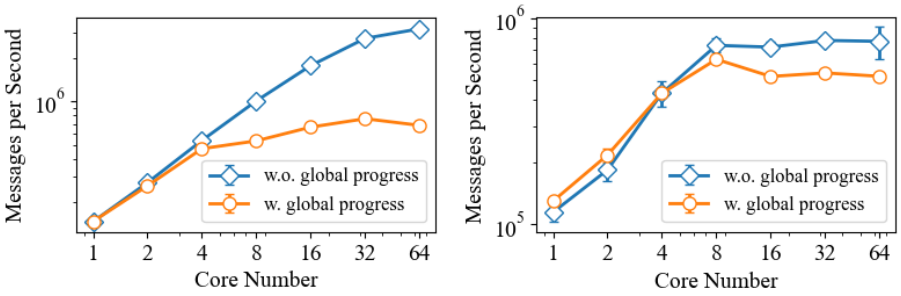


(b) Experiment Results on Delta.

Fig. 1: Performance impacts of the VCI extension compared to other MPI variants with 1-64 threads per process.

In addition to the VCI and the continuation extensions, the *mpix* results shown in Fig. 1 were measured with two special options: (1) the global progress was disabled with `MPIR_CVAR_CH4_GLOBAL_PROGRESS` set to 0, and (2) we disabled the usage of the continuation request by passing `MPI_REQUEST_NULL` as the *cont_request* argument to the `MPIX_Continue` function. We discuss these two additions and their performance impact in the next two sections.

4.3 Global Progress with Multiple VCIs



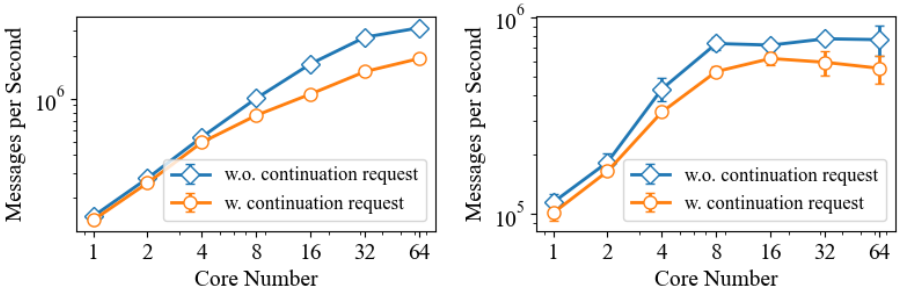
(a) Expanse. (b) Delta.

Fig. 2: Performance impacts of the global progress requirement with 1-64 threads per process.

As discussed in Section 3.2, MPICH employs an occasional global progress strategy by default for stronger MPI progress semantics, at the cost of increased thread contention across VCIs. Figure 2 shows the performance impact. We evaluate two variants (configured by the `MPIR_CVAR_CH4_GLOBAL_PROGRESS` control variable): one with occasional global progress enabled (the default option) and the other with it disabled (the option used by HPX). We observe that performance significantly improves when we disable the global progress option, even though it only performs one global progress every 255 per-VCI progress tests. The message rate improves by 4.5x on Expanse and 40% on Delta.

4.4 Continuation with Multiple Threads

As discussed in Section 3.3, the continuation request gives users more control over the progress and completion of pending MPI operations but also adds overhead. Figure 3 shows its performance impact. We evaluate two variants with/without the continuation requests. The variant with the continuation request allocates one continuation request per VCI, so there will be no contention on the VCI progress engines. The performance is improved when we disable the continuation request (by passing `MPI_REQUEST_NULL` as the `cont_request` argument to the



(a) Expanse. (b) Delta.

Fig. 3: Performance impacts of the continuation request with 1-64 threads per process.

`MPIX_Continue` function). With 64 threads, the performance improves by 64% on Expanse and 27% on Delta. This indicates that the atomic operations and the logic for completing and restarting continuation requests cause a noticeable overhead.

4.5 Summary

The VCI extension greatly improves the maximum message rate achievable in multithreaded scenarios. However, the two existing network backends in MPICH (UCX and OFI) have limitations. The global progress requirement of the MPI specification and the continuation request construct of the existing continuation proposal can also hurt the performance.

5 HPX Evaluation

In this section, we evaluate the performance impacts of the VCI and continuation extensions on the MPIx parcelport. We do this using two major benchmarks: an HPX microbenchmark, with a flood of messages between two nodes, which tests the maximum throughput of message processing; and an astrophysics application, OctoTiger [14], which tests the impact on a real-world application. [42] describes in detail the HPX flooding microbenchmark.

For the HPX microbenchmark, we report the achieved message rate for two payload sizes: 8 bytes and 16 kilobytes. With 8-byte payloads, the header message can piggyback the application data, and every parcel uses one MPI message. With 16-kilobyte payloads, every parcel uses two MPI messages: one header message and one data message. For the OctoTiger benchmark, we show the total execution time of the application with 20 iterations on 32 nodes. We run two OctoTiger processes per node and 63 threads per process, reserving 1 CPU core per socket for OS activities.

We also include the performance number of the LCI parcelport [41] for comparison. LCI [43] is an experimental communication library specifically designed for efficient asynchronous and multithreaded communication. Its interface and runtime are designed with AMT in mind, enabling a more direct communication path between the network and application layers. We include the LCI parcelport as a reference point representing achievable performance when programming directly against the native network API, unconstrained by the MPI standard.

5.1 Overall Performance

We first compare the MPIx parcelport (*mpix*) with the existing LCI parcelport (*lci*) and the original MPI parcelport (*mpi*) in HPX.

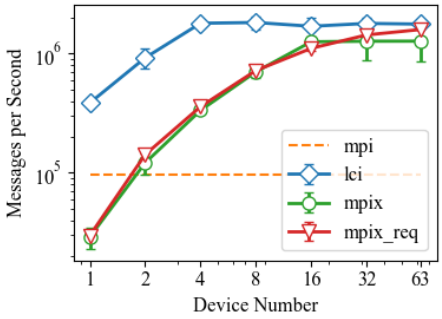
Fig. 4 shows the experimental results with varying numbers of MPICH VCIs or LCI devices. We show the results of the LCI parcelport (*lci*), the MPIx parcelport with continuation (*mpix*), the MPIx parcelport with request polling (*mpix_req*), and the old MPI parcelport (*mpi*). We observe that *mpix* greatly shrinks the performance gap between *lci* and *mpi*, especially on Expanse at higher device counts. *lci* is expected to outperform MPI-based approaches as it features a full redesign from the network layer up to suit AMT needs. The performance of *mpi* is much worse than that of *mpix*, showing the performance benefit of the VCI extensions.

Continuation-based programming simplifies development compared to managing and polling request pools, offering clear programmability benefits. It also yields a 5% performance improvement for OctoTiger on Expanse but shows no measurable gains in other scenarios, contrary to our expectations. This suggests that the overhead of request polling is less significant than anticipated.

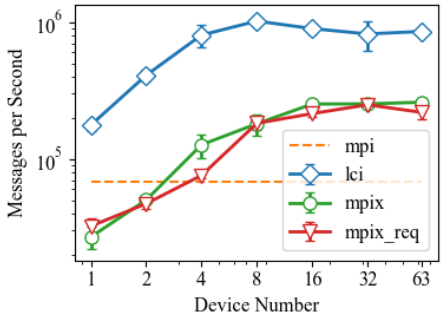
A prior study of the LCI parcelport [42] observed that a lightweight polling mechanism is indeed beneficial compared to the request polling mechanism, seemingly contradicting the observation here. However, LCI has a more thread-efficient runtime than MPICH, as LCI uses atomic-based data structures while MPICH uses a per-VCI spinlock to ensure thread safety. As a result, the contention due to lock-based request polling is relatively more significant with the LCI parcelport; this effect is hidden in the *mpix* case as MPICH's per-VCI spinlock is already coarse-grained. As a result, we believe the continuation extension will be beneficial when the MPICH runtime gets rid of the coarse-grained per-VCI spinlock and uses a more efficient lock-free data structure.

5.2 Investigate the Slowdown with Too Many VCIs

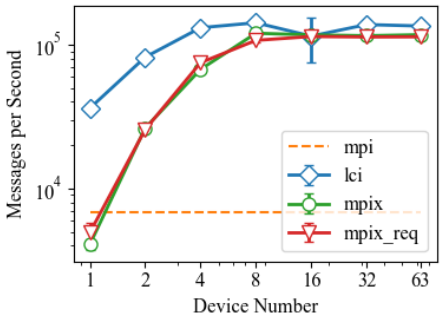
It is commonly believed that using one VCI per thread yields optimal multithreaded performance. However, our results show that using too many VCIs can degrade performance in real-world applications. We have observed this in the Octo-Tiger benchmark, where performance deteriorates with more than 16 VCIs. We also see a similar upward curve with the LCI parcelport. We further investigated why too many MPICH VCIs or LCI devices worsen performance. We identify **the attentiveness problem** as the main reason.



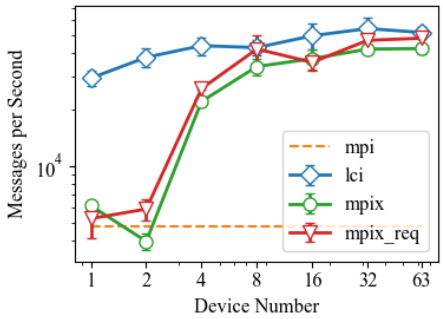
(a) Message Rate (8B) achieved with the flooding microbenchmark on Expanse.



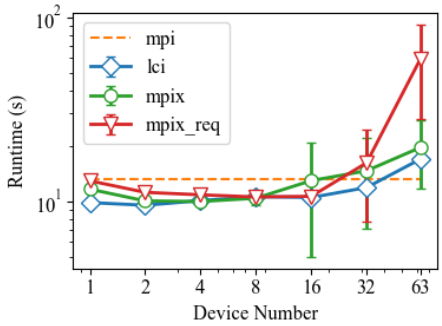
(b) Message Rate (8B) achieved with the flooding microbenchmark on Delta.



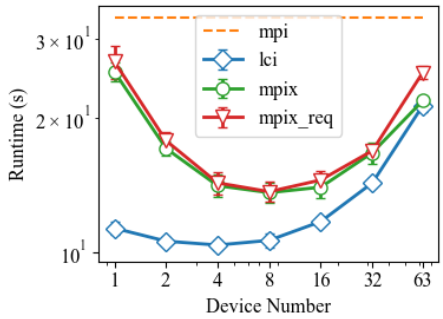
(c) Message Rate (16KiB) achieved with the flooding microbenchmark on Expanse.



(d) Message Rate (16KiB) achieved with the flooding microbenchmark on Delta.



(e) Octo-Tiger time per step with 32 nodes on Expanse.

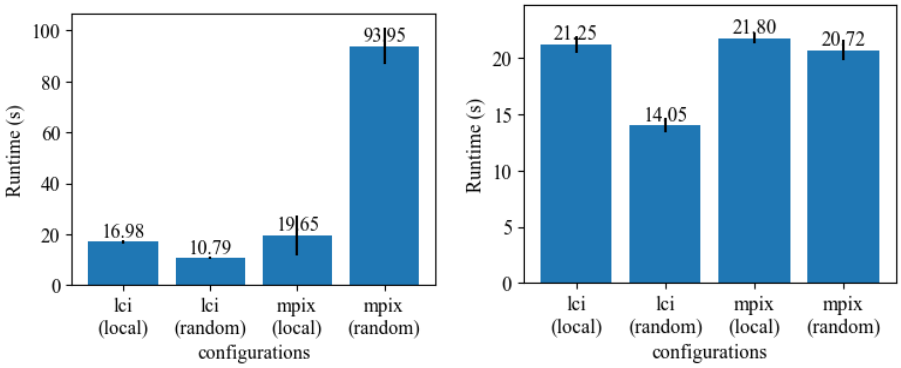


(f) Octo-Tiger time per step with 32 nodes on Delta.

Fig. 4: Performance impacts of using multiple VCIs and continuation. *lci* uses LCI. *mpi* uses the original MPI parcelport. *mpix* uses MPICH with the VCI and continuation extensions.

With too many VCIs, each VCI may not get enough attention from the threads. For 63 threads and 63 VCIs, each VCI only gets one thread to poll it. If the thread gets stuck executing a long-running task, it will not poll the corresponding VCI, and pending communications on that VCI will not be processed, even though other threads may be idle and waiting for work. With fewer VCIs, more threads poll each VCI, and pending communications on that VCI will be processed more quickly. On the other hand, there is more contention.

We implement a new progress strategy (*random*) in the MPIx parcelport and the LCI parcelport to verify this hypothesis and explore a potential fix. In the *random* strategy, each thread randomly picks a VCI to poll from all available VCIs. This way, even if a thread gets stuck executing a long-running task, other threads can still progress the pending communications on that VCI. Correspondingly, we name the previous strategy as *local*, as each thread only polls its own VCI.



(a) Expanse. (b) Delta.

Fig. 5: Performance impacts of the *random* progress strategies on OctoTiger execution time with 63 threads per process and 1 VCI per thread.

Figure 5 shows the performance impact of the *random* progress strategy. We notice that it greatly improves the performance of the LCI parcelport. However, it does not improve the performance of the MPIx parcelport. Instead, it worsens it on Expanse. This is due to the different threading efficiency of the two communication runtimes. In MPICH, every progress call will block waiting for the per-VCI spinlock, while in LCI, the progress call is non-blocking and always employs a try-lock wrapper around the low-level network resources [43]. Profiling confirms that MPICH gets stuck in the VCI spinlock more often with the *random* strategy.

5.3 Summary

The VCI extension shows great performance benefits across the HPX microbenchmark and the real-world application. The continuation extension does not show much performance benefits compared to per-VCI request polling with the current MPICH implementation. The recommended usage of one VCI per thread may not work with real-world applications where tasks both compute and communicate due to the attentiveness problem.

6 Related Work

Multiple efforts have sought to improve MPI performance in multithreaded environments. Prior work [5, 17, 3, 2, 34] has focused on reducing lock contention and minimizing the scope of critical sections within the MPI runtime. Other approaches [22, 25, 11, 21] leverage user-level threads, task systems, or process-in-process techniques to enhance MPI on many-core processors and irregular workloads. More recently, [45, 46] proposed using VCI to replicate low-level network resources, thereby removing the need for runtime-level serialization. The VCI mechanism has since become the recommended approach for improving multithreaded performance in MPICH, representing a major milestone in MPI implementation-level optimization. Similar optimizations have also been adopted in OpenMPI [34].

A complementary line of research has focused on extending the MPI interface to support multithreaded execution better. [16] proposed the endpoints extension, which decouples threads from ranks and enables threads within a process to issue non-contending MPI calls with different endpoints. More recent MPIX Stream [47] and thread communicator [48] extensions revive and refine the endpoint model. These interface-level extensions help users better convey thread-level parallelism to the MPI runtime. Under the hood, the MPI runtime still relies on VCIs for better multithreaded performance.

Beyond the MPI ecosystem, several other communication libraries have been developed to support asynchronous and multithreaded communication. GASNet and GASNet-EX [10, 9] provide low-level active messages and RMA operations for library developers and compiler-generated codes. At a higher level, PGAS models like UPC [18], UPC++ [4], and OpenSHMEM [12] expose global memory abstractions with one-sided RMA operations. LCI [43] proposes new interface and runtime designs to enhance multithreaded communication performance and simplify asynchronous programming.

In contrast to these efforts, our work does not propose new MPI extensions or communication libraries. Instead, we focus on evaluating the practical effectiveness of existing mechanisms, specifically the VCI and continuation extensions, and identifying their limitations. Our analysis complements prior work, offering detailed insights into how current MPI features can be better utilized and where future improvements are needed.

7 Conclusion and Discussion

In this paper, we evaluated the effectiveness of the VCI and continuation extensions in MPICH using both MPI-level and HPX-level benchmarks. Our results show that the VCI extension can significantly improve the performance of multithreaded applications. The continuation extension, while beneficial for programmability, currently shows limited performance benefit.

Contrary to the common recommendation of assigning one VCI per thread, we found that excessive use of VCIs can degrade performance in real-world applications. We identified the attentiveness problem as the primary cause: when too many VCIs are in use, the MPI runtime may fail to poll them frequently enough, leading to increased latency and missed progress opportunities. Our findings highlight intra-VCI threading efficiency as a critical factor. Improving it not only resolves the attentiveness issue by enabling more efficient polling across threads, but also allows users to meet their multithreaded communication needs with fewer VCIs, which will also boost scalability by reducing resource usage.

Improved intra-VCI efficiency also helps demonstrate the benefits of the continuation extension. Continuations eliminate the need for explicit polling of shared request pools, thus removing the associated thread contention. However, if intra-VCI operations rely on coarse-grained locks, internal contention can obscure these gains. With more efficient intra-VCI handling, continuations can better realize their potential of minimizing overhead and avoiding contention.

While it is known to be challenging to design a threading-efficient VCI due to the non-overtaking requirement and the need to support wildcard receives, recent MPI info keys such as *allow_overtaking* and *no_any_tag/source* offer a practical path forward. When these keys are set, MPI runtimes can safely adopt more scalable designs. Task systems, some of the primary users of asynchronous multithreaded communication, can often tolerate these relaxations [42]. However, they still require support for *any_source* receives, which may necessitate additional info keys. One possible approach is to propagate *any_source* information to the sender side, as suggested in prior work [15].

In addition, our evaluation has revealed limitations in two commonly used communication middlewares: UCX and libfabric. Specifically, UCX shows performance degradation when more than 16 UCP workers are used, and libfabric delivers lower absolute performance. Prior work on LCI [43] has demonstrated that multithreaded performance comparable to MPI-everywhere (one process per core) is achievable when building directly on top of the libibverbs [32] layer. Addressing these performance constraints in the underlying middleware is essential for MPI implementations to fully realize scalable multithreaded communication.

We believe these insights offer practical guidance for improving multithreaded communication performance in MPICH and other MPI implementations, and we hope they inform future runtime and interface design.

Acknowledgements. This work used Expanse at San Diego Supercomputer Center [39] and Delta at National Center for Supercomputing Applications [19]

through allocations CCR130058 and CIS250465 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [8], which is supported by U.S. National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

1. Abdulah, S., Baker, A. H., Bosilca, G., Cao, Q., Castruccio, S., Genton, M. G., Keyes, D. E., Khalid, Z., Ltaief, H., Song, Y., Stenchikov, G. L., and Sun, Y. Boosting earth system model outputs and saving petabytes in their storage using exascale climate emulators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2024), SC '24, IEEE Press.
2. Amer, A., Lu, H., Balaji, P., Chabbi, M., Wei, Y., Hammond, J., and Matsuoka, S. Lock contention management in multithreaded MPI. *ACM Transactions on Parallel Computing 5*, 3 (2019-01-08), 12:1–12:21.
3. Amer, A., Lu, H., Wei, Y., Balaji, P., and Matsuoka, S. MPI+threads: Runtime contention and remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015-01-24), PPoPP 2015, Association for Computing Machinery, pp. 239–248.
4. Bachan, J., Baden, S. B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P. H., and Ahmed, H. UPC++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), pp. 963–973.
5. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., and Thakur, R. Toward efficient support for multithreaded MPI communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2008), A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds., Lecture Notes in Computer Science, Springer, pp. 120–129.
6. Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–11.
7. Bernholdt, D. E., Boehm, S., Bosilca, G., Gorentla Venkata, M., Grant, R. E., Naughton, T., Pritchard, H. P., Schulz, M., and Vallee, G. R. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience 32*, 3 (2020), e4851. e4851 cpe.4851.
8. Boerner, T. J., Deems, S., Furlani, T. R., Knuth, S. L., and Towns, J. Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*. 2023, pp. 173–176.
9. Bonachea, D., and Hargrove, P. H. GASNet-EX: A high-performance, portable communication library for Exascale. In *Languages and Compilers for Parallel Computing: 31st International Workshop (LCPC 2018)* (2018), Springer, pp. 138–158.
10. Bonachea, D., and Jeong, J. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring 31* (2002), 17.

11. Carribault, P., Pérache, M., and Jourdren, H. Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In *International Workshop on OpenMP* (2010), Springer, pp. 1–14.
12. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., and Smith, L. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, NY, USA, 2010), PGAS '10, Association for Computing Machinery.
13. Chatterjee, S., Tasirlar, S., Budimlic, Z., Cavé, V., Chabbi, M., Grossman, M., Sarkar, V., and Yan, Y. Integrating asynchronous task parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), IEEE, pp. 712–725.
14. Daiss, G., Diehl, P., Yan, J., Holmen, J. K., Gayatri, R., Junghans, C., Straub, A., Hammond, J. R., Marcello, D., Tsuji, M., et al. Asynchronous-many-task systems: Challenges and opportunities–scaling an amr astrophysics code on exascale machines using kokkos and hpx. *arXiv preprint arXiv:2412.15518* (2024).
15. Dang, H.-V., Snir, M., and Gropp, W. Towards millions of communicating threads. In *Proceedings of the 23rd European MPI Users' Group Meeting* (New York, NY, USA, 2016), EuroMPI '16, Association for Computing Machinery, p. 1–14.
16. Dinan, J., Grant, R. E., Balaji, P., Goodell, D., Miller, D., Snir, M., and Thakur, R. Enabling communication concurrency through flexible mpi endpoints. *The International Journal of High Performance Computing Applications 28*, 4 (2014), 390–405.
17. Dózsa, G., Kumar, S., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Ratterman, J., and Thakur, R. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *Recent Advances in the Message Passing Interface* (2010), R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds., Lecture Notes in Computer Science, Springer, pp. 11–20.
18. El-Ghazawi, T., and Smith, L. UPC: Unified parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, Association for Computing Machinery, p. 27–es.
19. Gropp, W., Boerner, T., Bode, B., and Bauer, G. Delta: Balancing gpu performance with advanced system interfaces.
20. Hofmeyr, S., Egan, R., Georganas, E., Copeland, A. C., Riley, R., Clum, A., Eloe-Fadrosh, E., Roux, S., Goltsman, E., Buluç, A., et al. Terabase-scale metagenome coassembly with metahipmer. *Scientific reports 10*, 1 (2020), 10689.
21. Hori, A., Si, M., Gerofi, B., Takagi, M., Dayal, J., Balaji, P., and Ishikawa, Y. Process-in-process: techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2018), HPDC '18, Association for Computing Machinery, p. 131–143.
22. Huang, C., Lawlor, O., and Kale, L. V. Adaptive MPI. In *International workshop on languages and compilers for parallel computing* (2003), Springer, pp. 306–322.
23. Kaiser, H., et al. HPX - The C++ standard library for parallelism and concurrency. *Journal of Open Source Software 5*, 53 (2020), 2352.
24. Kale, L. V., and Krishnan, S. CHARM++: A portable concurrent object oriented system based on C++. 91–108.

25. Kamal, H., and Wagner, A. FG-MPI: Fine-grain MPI for multicore and clusters. In *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)* (2010), pp. 1–8.
26. Ltaief, H., Alomairy, R., Cao, Q., Ren, J., Slim, L., Kurth, T., Dorschner, B., Bougouffa, S., Abdelkhalak, R., and Keyes, D. E. Toward capturing genetic epistasis from multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis* (2024), pp. 1–12.
27. Mei, C., Sun, Y., Zheng, G., Bohm, E. J., Kale, L. V., Phillips, J. C., and Harrison, C. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–11.
28. Mor, O., Bosilca, G., and Snir, M. Improving the scaling of an asynchronous many-task runtime with a lightweight communication engine. In *Proceedings of the 52nd International Conference on Parallel Processing* (2023), ICPP '23, Association for Computing Machinery, pp. 153–162.
29. Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, Association for Computing Machinery, p. 103–112.
30. MPI Forum. MPI: a message passing interface standard, Nov. 2023.
31. MPICH Developers. MPICH: High-Performance Portable MPI. https://www.mpich.org, n.d.
32. NVIDIA. Rdma aware networks programming user manual, 2025.
33. (OFIWG), O. W. G. Libfabric programmer's manual, 2023.
34. Patinyasakdikul, T., Eberius, D., Bosilca, G., and Hjelm, N. Give MPI threading a fair chance: A study of multithreaded MPI designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)* (2019), pp. 1–11.
35. Patinyasakdikul, T., Luo, X., Eberius, D., and Bosilca, G. Multirate: A flexible mpi benchmark for fast assessment of multithreaded communication performance. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)* (2019), pp. 1–11.
36. Schuchart, J. MPI continuations proposal, 2021.
37. Schuchart, J., Samfass, P., Niethammer, C., Gracia, J., and Bosilca, G. Callback-based completion notification using MPI continuations. *Parallel Computing 106* (2021-09-01), 102793.
38. Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., and Bouteiller, A. UCX: An open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects* (2015), pp. 40–43.
39. Strande, S., Cai, H., Tatineni, M., Pfeiffer, W., Irving, C., Majumdar, A., Mishin, D., Sinkovits, R., Norman, M., Wolter, N., Cooper, T., Altintas, I., Kandes, M., Perez, I., Shantharam, M., Thomas, M., Sivagnanam, S., and Hutton, T. Expanse: Computing without boundaries: Architecture, deployment, and early operations experiences of a supercomputer

designed for the rapid evolution in science and engineering. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions* (New York, NY, USA, 2021), PEARC '21, Association for Computing Machinery.

40. Yadav, R., Lee, W., Elibol, M., Papadakis, M., Lee-Patti, T., Garland, M., Aiken, A., Kjolstad, F., and Bauer, M. Legate sparse: Distributed sparse computing in python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2023), SC '23, Association for Computing Machinery.
41. Yan, J., Kaiser, H., and Snir, M. Design and analysis of the network software stack of an asynchronous many-task system – the LCI parcelport of HPX. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (New York, NY, USA, 2023), SC-W '23, Association for Computing Machinery, p. 1151–1161.
42. Yan, J., Kaiser, H., and Snir, M. Understanding the communication needs of asynchronous many-task systems–a case study of HPX+LCI. *arXiv preprint arXiv:2503.12774* (2025).
43. Yan, J., and Snir, M. Lci: a lightweight communication interface for efficient asynchronous multithreaded communication. *arXiv preprint arXiv:2505.01864* (2025).
44. Zambre, R., Chandramowliswharan, A., and Balaji, P. How i learned to stop worrying about user-visible endpoints and love MPI. In *Proceedings of the 34th ACM International Conference on Supercomputing* (New York, NY, USA, 2020), ICS '20, Association for Computing Machinery.
45. Zambre, R., Chandramowliswharan, A., and Balaji, P. How I learned to stop worrying about user-visible endpoints and love MPI. In *Proceedings of the 34th ACM International Conference on Supercomputing* (2020), ICS '20, Association for Computing Machinery, pp. 1–13.
46. Zambre, R., Sahasrabudhe, D., Zhou, H., Berzins, M., Chandramowlishwaran, A., and Balaji, P. Logically Parallel Communication for Fast MPI+Threads Applications. 3038–3052.
47. Zhou, H., Raffenetti, K., Guo, Y., and Thakur, R. MPIX Stream: An explicit solution to hybrid MPI+X programming. In *Proceedings of the 29th European MPI Users' Group Meeting* (2022), pp. 1–10.
48. Zhou, H., Raffenetti, K., Zhang, J., Guo, Y., and Thakur, R. Frustrated with MPI+Threads? try MPIxThreads! In *Proceedings of the 30th European MPI Users' Group Meeting* (2023), pp. 1–10.