How I Learned to Stop Worrying about User-Visible Endpoints and Love MPI

Rohit Zambre
rzambre@uci.edu
University of California, Irvine

Aparna
Chandramowliswharan
amowli@uci.edu
University of California, Irvine

Pavan Balaji
balaji@anl.gov
Argonne National Laboratory

ABSTRACT

MPI+threads is gaining prominence as an alternative to the traditional "MPI everywhere" model in order to better handle the disproportionate increase in the number of cores compared with other on-node resources. However, the communication performance of MPI+threads can be 100x slower than that of MPI everywhere. Both MPI users and developers are to blame for this slowdown. MPI users traditionally have not exposed logical communication parallelism. Consequently, MPI libraries have used conservative approaches, such as a global critical section, to maintain MPI's ordering constraints for MPI+threads, thus serializing access to the underlying parallel network resources and limiting performance.

To enhance the communication performance of MPI+threads, researchers have proposed MPI Endpoints as a user-visible extension to the MPI-3.1 standard. MPI Endpoints allows a single process to create multiple MPI ranks within a communicator. This could, in theory, allow each thread to have a dedicated communication path to the network, thus avoiding resource contention between threads and improving performance. The onus of mapping threads to endpoints, however, would then be on domain scientists. In this paper we play the role of devil's advocate and question the need for such user-visible endpoints. We certainly agree that dedicated communication channels are critical. To what extent, however, can we hide these channels inside the MPI library without modifying the MPI standard and thus unburden the user? More important, what functionality would we lose through such abstraction? This paper answers these questions through a new implementation of the MPI-3.1 standard that uses multiple virtual communication interfaces (VCIs) inside the MPI library. VCIs abstract underlying network contexts. When users expose parallelism through existing MPI mechanisms, the MPI library maps that parallelism to the VCIs, relieving the domain scientists from worrying about endpoints. We identify cases where user-exposed parallelism on VCIs perform as well as user-visible endpoints, as well as cases where such abstraction hurts performance.

CCS CONCEPTS

• **Software and its engineering → Massively parallel systems**; *Multithreading*; Message oriented middleware.

KEYWORDS

MPI+threads, MPI+OpenMP, MPI_THREAD_MULTIPLE, exascale MPI, high-performance communication, MPI Endpoints

1 INTRODUCTION

MPI everywhere (typically one MPI process per core) has been the traditional model for using MPI on supercomputers. While the model has served applications well for several decades, it is becoming difficult to scale on modern architectures, primarily owing to the disproportionate increase in the number of cores per node compared with other on-node resources such as memory and network registers. For example, memory wastage for halo regions in PDE simulations with the MPI everywhere model is a known problem [35], which worsens with the increase in dimensionality of the domain decomposition. To address this issue, researchers have been increasingly adopting hybrid MPI+threads (typically one MPI process per node or socket, and one thread per core) parallelism (e.g., MPI+OpenMP) [18] since it allows them to utilize the many cores on a node while sharing the remaining on-node resources [19, 27, 30].

The communication performance of MPI+threads, however, is dismal, especially when multiple threads are involved in communication (i.e., MPI_THREAD_MULTIPLE). The reason for the poor performance stems from the quaint view, held by both MPI users and MPI developers, of the network as a sequential hardware resource. Modern network interface cards (NICs) feature multiple hardware communication contexts that allow for independent, parallel communication streams from a single node [42]. To efficiently utilize the network parallelism, MPI users must expose logical parallelism in their communication so that threads can map to different underlying hardware contexts on the NIC. How does one expose such logical communication parallelism? The MPI standard specifies certain sequential ordering constraints between messages [9] that guarantee some determinism in execution. But these sequential ordering constraints are based on <communicator, rank, tag> or <window, rank> tuples. The user can inform the MPI library that two or more messages have no relative ordering between them by using, for example, different communicators, different windows, or in some cases different ranks or tags, thus exposing logical parallelism between these messages.

The state of the art, however, is conservative in this regard. Applications typically do not expose MPI communication parallelism because MPI libraries today do not utilize such parallelism. MPI libraries, on the other hand, still employ conservative approaches, such as a global critical section and the use of only one network hardware context, because applications today do not expose any parallelism that the library can exploit.

To facilitate the improvement in the communication performance of MPI+threads, user-visible MPI Endpoints [8, 20] has been proposed as an extension to the MPI-3.1 standard. These user-visible endpoints allow the user to explicitly map threads to the underlying network resources. If the user mapped each thread to a distinct endpoint, then, in theory, all threads could have dedicated communication paths to the network. Several efforts [22, 29, 40] indeed demonstrate scaling communication throughput with MPI Endpoints through dedicated communication channels inside the MPI library. Unfortunately, these works have a number of shortcomings. First and foremost, user-visible solutions enforce the new concept of endpoints upon users through new APIs and put the onus of mapping threads to the endpoints on the domain scientist, potentially hurting productivity. Second, prior works did not compare against an MPI-3.1 implementation that uses multiple network hardware contexts, thus implicitly assuming that current implementations of MPI-3.1 are already the most optimal. Third, they modify applications to expose communication parallelism to the MPI library with MPI Endpoints but do not expose the equivalent parallelism to the MPI-3.1 version of the library, thus making the comparison unfair. Finally, irrespective of whether we optimize the implementation of MPI-3.1 or implement MPI Endpoints, certain corner cases with respect to communication progress must be handled for correctness, even though they sometimes hurt performance. Prior works ignore such corner cases, sacrificing correctness in pursuit of higher performance.

In this paper, we play the role of devil's advocate to user-visible endpoints. In fairness to the previous efforts, we agree with them on two aspects: (1) applications must expose communication parallelism, but MPI-3.1 already provides multiple mechanisms to do that; and (2) MPI libraries must provide multiple independent communication channels, but an efficient MPI library can do so internally without exposing them to users as endpoints. Thus we are left with the question: Are extensions to the MPI standard necessary in order to improve MPI+threads communication?

To answer this question, we start with a new MPI-3.1 library that internally uses multiple virtual communication interfaces (VCIs). A VCI represents a communication stream that is mapped to a network hardware context. Users expose communication parallelism through existing MPI mechanisms (such as communicators, windows, ranks, and tags), and the MPI library maps that parallelism to the different hardware contexts by funneling messages over the internal VCIs. More important, VCIs are completely hidden within the MPI library, thus requiring no extension to the MPI standard and placing no requirement for thread-to-network-resource mapping on the domain scientists.

The effectiveness of transparently using multiple VCIs depends on the communication pattern of applications. In this regard, we classify MPI+threads applications into three categories: (1) applications that can directly use dedicated communication channels where the multi-VCI approach saturates the network performance similarly to MPI everywhere and user-visible endpoints; (2) applications that require shared progress where both multiple VCIs and user-visible endpoints suffer from loss in performance; and (3) applications that need direct access to the network resources where abstracting the VCIs can hurt performance compared with user-visible endpoints. We study applications in all three categories in this paper. To that end, we make the following contributions:

(1) We develop a fast MPI+threads library by addressing thread safety and network resource underutilization while adhering to the MPI standard (in Section 4).
(2) We compare the capabilities of MPI-3.1 with those of user-visible endpoints for microbenchmarks and real applications (in Sections 5 and 6).
(3) We provide users with recommendations on exposing communication parallelism in their applications with MPI-3.1 (in Section 6) based on Section 2's discussion of parallelism in the existing MPI standard.

2 PARALLELISM IN THE MPI STANDARD

For both two- and one-sided MPI communication, the existing MPI standard allows applications to expose communication parallelism. Below, we discuss approaches for a single MPI process to expose such parallelism. In particular, communication parallelism in MPI can be viewed as multiple independent communication streams, where each stream is a first-in, first-out (FIFO) ordered set of communication operations.

2.1 Point-to-point communication

For two-sided communication, MPI uses the <communicator, rank, tag> triplet to match operations.

Different communicators. MPI does not define any order between operations executed on different communicators. This approach implies that all operations on different communicators can execute independently on parallel communication streams.

Same communicator, different ranks. Within a communicator, MPI specifies a *nonovertaking order* [9]: if multiple ordered operations match the same target operation, the operation that was issued first must consume the target operation before the one that was issued later. No matching order applies to operations intended for different targets. For example, no ordering constraints apply to multiple send operations that use the same communicator but target different ranks. Hence, they can execute on parallel communication streams. On the other hand, receive operations that use the same communicator cannot execute in parallel even if they specify different ranks. The reason is that it is possible for any receive operation to contain the MPI_ANY_SOURCE wildcard. To ensure correct matching order, the MPI library needs to funnel all receive operations of a communicator through the same communication stream (see Figure 1).

Same communicator, same rank, different tags. Operations that target the same rank within a communicator but use different tags cannot utilize parallel communication streams for both send and receive operations. The order of operations in MPI is determined by the MPI user. In MPI+threads, operations on different threads may be parallel or ordered through, for example, a thread
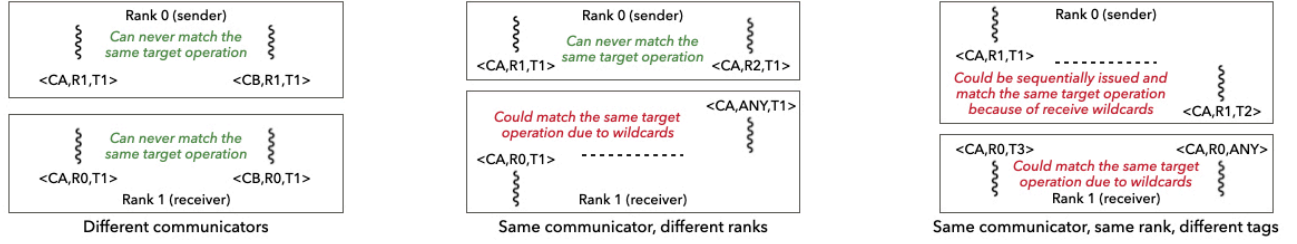
Figure 1: Different combinations of <comm,rank,tag> tuples demonstrating point-to-point parallelism in the MPI standard. Dashed horizontal lines represent thread barriers.

barrier. Suppose the user issues two operations on two different threads with a barrier between the operations (see Figure 1). A target operation that satisfies both operations must first match the operation that was issued before the barrier. To ensure this, the MPI library must use the same communication stream for the operations from the two threads. If the operations use different communication streams, the operation issued after the barrier could incorrectly match the target prior to the one issued before the barrier.

2.2 Remote Memory Access communication

MPI's one-sided communication, namely, remote memory access (RMA), is executed on top of windows. Unlike point-to-point, RMA operations do not have any matching constraints and feature a lot more parallelism. MPI does not require any ordering for its Get, Put, and Accumulate classes of operations if two or more operations target different ranks or use different windows. Additionally, two or more Put or Get operations do not have any ordering constraints even if they use the same window. Hence, multiple Get and Put types of operations can execute on parallel communication streams. But, by default, MPI-3.1 requires program order for Accumulate operations originating from the same source and targeting the same memory location on the same window. It does, however, give the user the option to relax this ordering constraint through the accumulate_ordering hint. Without hints, multiple Accumulate-style operations can execute on parallel communication streams if they use different windows or target different memory locations.

Even though multiple RMA operations on the same window could use parallel communication streams, mixing synchronization operations, such as MPI_Win_flush, with communication operations, such as MPI_Get, can be tricky. Synchronization calls can wait for both past and concurrent communication operations to complete. Thus, if one thread is waiting inside MPI_Win_flush and another thread continuously issues MPI_Get operations, the first thread might block indefinitely. Apart from these constraints, all types of RMA operations on different windows can execute through separate communication streams in parallel.

3 SOFTWARE AND TESTBEDS

Our MPI implementation is based on the highly optimized CH4 [36] device of the MPICH library. The CH4 device is a combination of three components: a core (ch4_core), a network module (netmod), and a shared-memory module (shmmod). The netmod and shmmod are responsible for conducting internode and intranode communication, respectively. In this work, we focus on the netmod component because we assume MPI+threads applications would directly use the shared memory of the process for intranode communication.

For most common data operations, CH4 offloads functionalities, such as tag matching, to the low-level communication library, such as OpenFabrics Interfaces (OFI) [26] or Unified Communication X (UCX) [38]. Where the hardware cannot independently handle operations, CH4 falls back on using an active message implementation of the operation in its ch4_core.

Our testbeds include two platforms: the Skylake cluster and the Gomez cluster in the Joint Laboratory for System Evaluation at Argonne National Laboratory. The clusters feature different interconnects: Skylake hosts Intel Omni-Path (OPA) and Gomez hosts Mellanox InfiniBand (IB) EDR. These two families of interconnects constitute the majority of the TOP500 in the supercomputing space [11]. For Skylake, we use the OFI netmod in conjunction with PSM2; for Gomez, we use the UCX netmod with Verbs.

For our analysis and evaluation, we use the cores on the socket that the NIC is attached to. We ensure that the CPU speed is set to its base frequency and that turbo boost is turned off.

4 A FAST MPI+THREADS LIBRARY

In this section, we detail our implementation of parallel communication streams (or VCIs) within a single MPI process. Although our work is on MPICH, the concepts extend to other MPI libraries as well. To design a fast MPI+threads library, we need to deserialize access to the software as well as to the network hardware resources; the former is a critical precursor for the latter to extract performance.

4.1 Deserializing access to the MPI library

In MPI+threads, the MPI library needs to protect its resources from the threads' parallel updates. State-of-the-art MPI implementations conservatively employ a large global critical section with a single lock. The MPI operation enters the critical section at the beginning of its execution and exits it either when it returns from the function or when it yields to other threads to make progress. This approach largely serializes communication from multiple threads even if the communication operations issued by those threads are independent.

Fine-grained critical sections. Balaji et al. [15, 16] and Amer et al. [12] split the global lock in MPICH into multiple locks such that each lock protects a different class of objects. For example,
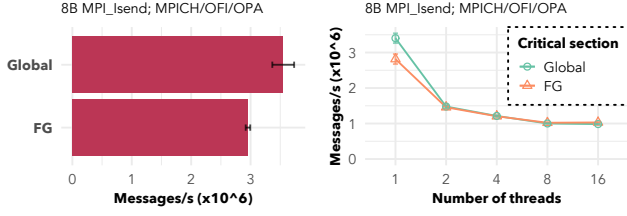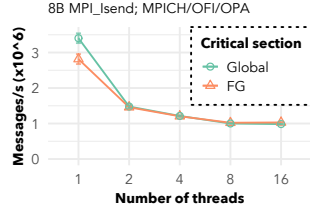
Figure 2: Overhead of FG.

Figure 3: Global vs. FG.

Figure 4: Multi-VCI `MPI_Init` and `MPI_Finalize` overheads.

access to the network communication portal is protected by a lock different from the one that protects the management of request objects. Although fine-grained critical sections (FG) mean higher parallelism, they incur two expenses over a global critical section (Global): (1) more lock acquisitions and releases on the critical path and (2) atomics for reference and completion counters.

The number of locks taken in FG depends on the type of operations. For any initiation operation, we need at least one lock—the one that protects access to the communication portal. Generally, for `MPI_Isend` and `MPI_Irecv`, we need a second lock—the one that allocates a request object from the global pool of requests. For small-message transmissions, however, we do not need the lock of the request pool. Up to a certain message size, modern interconnects guarantee completion as soon as they are posted; they do not require any polling of the network.[*] MPICH optimizes memory usage for such operations by maintaining a global lightweight request that is marked as complete. These operations then simply increase the reference counter of the pre-completed request.

For progress operations, the number of locks taken depends on the number of times the progress engine is invoked. The progress engine not only checks for the completion of an operation but also progresses active outstanding schedules, such as those of non-blocking collectives. One iteration of the progress engine in MPICH takes three locks: one to poll the communication portal and two to check the activeness of progress hooks[†] (each hook maintains its own thread safety). When an operation completes, another lock is taken when the request object is returned to the pool.

Although FG improves concurrency when multiple threads compete for MPI resources, it adds some overhead when there is no contention (e.g., when a single thread is active). Figure 2 shows that FG hurts performance by 16.71% in the uncontended case (compared with Global). This performance difference is due to the higher number of locks and to atomic counting (as we corroborate in Section 5). With increasing number of threads, the performance difference between FG and Global reduces, and FG eventually outperforms Global at 16 threads, as seen in Figure 3. Moreover, although Global performs better than FG for fewer threads, FG is critical when parallel communication streams exist, as we show in Section 4.3.

4.2 Parallel communication streams

To address the problem of network resource underutilization, we first define the virtual communication interface object. A VCI is an
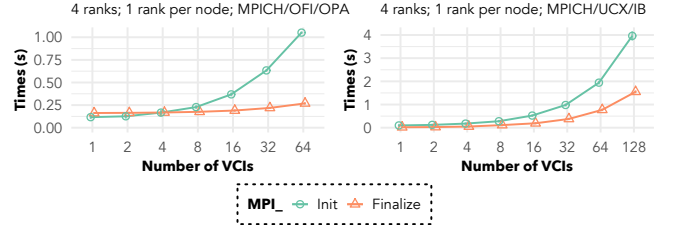
abstract representation of a communication stream. Each VCI maps to a communication context on the network hardware and contains its own independent set of communication resources that maintain a FIFO order of the MPI operations that map to it. Hence, with multiple VCIs, we get parallel communication streams in the MPI library. The physical realization of a VCI depends on the `netmod` and the underlying interconnect. A VCI in the OFI `netmod` is an OFI endpoint (for transmission and reception) that is bound to an OFI completion queue (for progress). For Intel OPA, the OFI endpoint maps to a hardware context on the Intel HFI network adapter [4]. A VCI in the UCX `netmod` is a UCP worker. For Mellanox IB, the UCP worker contains Verbs resources: a queue pair (QP) for transmission, a shared receive queue for reception, and a completion queue for progress. The QP maps to the micro UARs (hardware registers) on the Mellanox adapter [6].

VCI pool design. To utilize the underlying network parallelism, we maintain a pool of VCIs inside a single MPI process. Since operations on different communicators can execute on different communication streams (see Section 2), every time the user creates a new communicator, we assign it a VCI from the pool and mark that VCI as active. All operations on the communicator now funnel through the VCI that was assigned to the communicator. If multiple threads communicated using separate communicators, they would, in theory, establish parallel communication streams to the NIC from the same process. However, since the number of contexts on the network hardware is limited,[‡] the VCI pool may be empty during communicator creation. In such a case, we revert to a fallback VCI. For this work, we designate the VCI allocated to `MPI_COMM_WORLD` as the fallback. When the user frees a communicator, its associated VCI is returned to the pool and is marked as inactive. Certainly better techniques to map communicators to VCIs exist, but such techniques are out of the scope of this paper and will be analyzed in the future. The overhead from this design is that each operation now needs to compute which VCI to use on the critical path. For the communicator-to-VCI mapping, this computation is a lookup, which costs 8 additional instructions in our implementation. The VCI pool design extends to RMA operations as well, where we assign VCIs to each window since operations on different windows can execute in parallel (see Section 2).

Thread safety. We extend the fine-grained critical sections from Section 4.1 such that each VCI is then protected by its own separate lock since it is independent. Threads that map to different VCIs can access the VCIs without contention.

---

[*]A correct MPI implementation would need to poll the network intermittently even for such operations to progress any active message execution of an operation.
[†]MPICH/CH4 currently maintains two progress hooks.

[‡]For example, Intel OPA features only 160 hardware contexts on the HFI adapter [4]
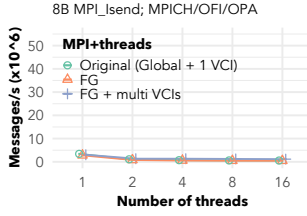
Figure 5: Multiple VCIs.

Figure 6: Progress opts.
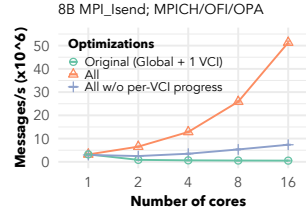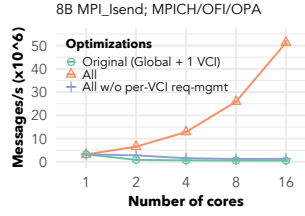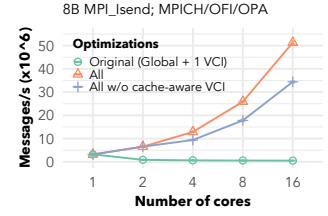

Figure 7: Request opts.

Figure 8: Cache-aware VCI.

```
1  /*Point-to-point example*/       1  /*RMA example (large Puts)*/
2  Rank 0:                          2  Rank 0:
3    MPI_Ssend(comm1);              3    MPI_Get(win1);
4    MPI_Ssend(comm2);              4    MPI_Get(win2);
5                                   5    MPI_Win_flush(win1);
6                                   6    MPI_Win_flush(win2);
7                                   7
8  Rank 1 / Thread 0:              8  Rank 1 / Thread 0:
9    MPI_Irecv(comm1,req1);        9    MPI_Get(win1);
10 #pragma omp barrier            10 #pragma omp barrier
11 #pragma omp barrier            11 #pragma omp barrier
12   MPI_Wait(req1);              12   MPI_Win_flush(win1);
13                                13
14 Rank 1 / Thread 1:            14 Rank 1 / Thread 1:
15   MPI_Irecv(comm2,req2);      15   MPI_Get(win2);
16 #pragma omp barrier            16 #pragma omp barrier
17   MPI_Wait(req2);              17   MPI_Win_flush(win2);
18 #pragma omp barrier            18 #pragma omp barrier
```

Figure 9: Point-to-point (left) and RMA (right) scenarios that would deadlock without shared progress of VCIs.

Connection establishment. Each VCI has its own transport-level address that needs to be exchanged between the ranks in order to establish connections. We do so during the initialization of MPI. We first use PMI [14] to exchange the addresses of the fallback VCIs on every rank. Using the fallback VCI, we exchange the addresses of the rest of the VCIs using an allgather operation. As expected, establishing connections statically during initialization incurs an overhead that grows with the number of VCIs (see Figure 4). Similarly, the finalization time increases since the tear-down time of VCIs is proportional to the number of VCIs.§

4.3 Optimizing multi-VCI communication

Figure 5 shows that simply using multiple VCIs produces practically no performance benefit. We present several optimizations to the multi-VCI communication introduced in Section 4.2.

Per-VCI progress. With only one VCI (Original), the job of the progress function was simple: poll for progress on the single VCI. With multiple VCIs, a naïve extension would be to poll for progress on all the active VCIs. Although correct, this approach would be detrimental to performance especially when multiple threads progress operations in parallel since they would be contending on the VCIs' locks. Also, each thread would be doing more work than necessary. Because all MPI communication operations map to a VCI, progress for an operation primarily needs to poll the

VCI on which the operation was posted. We extend the progress engine to allow for *per-VCI progress*. First, we store the VCI used for an operation in its request object. This action adds 3 instructions to the critical path. Using the information stored in the request object, the progress functions poll for progress on the VCI that was used for the operation. When multiple threads progress operations mapped to different VCIs, they do not contend.

Although per-VCI progress helps improve performance, progressing only the VCI used by the current request is incorrect and can lead to deadlock. Consider the point-to-point example in Figure 9. This is a correct MPI program—the first synchronous send¶ on rank 0 (line 3) should return because its matching receive has already been posted (line 9). With current MPI libraries, this program completes because `MPI_Wait(req2)` (line 17) initiates the reception of `MPI_Ssend(comm1)` by polling the single VCI that both communicators map to, thus allowing `MPI_Ssend(comm1)` to return. With multiple VCIs and per-VCI progress, `MPI_Wait(req2)` progresses only the VCI associated with `comm2`, preventing `MPI_Ssend(comm1)` to complete and causing deadlock. Figure 9 also describes a similar scenario with RMA operations using passive-target synchronization for cases where the underlying network requires target-side CPU involvement for progress.

In summary, the pure per-VCI progress model can improve performance, but the global progress model is necessary to ensure correctness even though it loses some performance. To account for such communication patterns, we use a hybrid progress model; that is, we perform one round of global progress after a certain number of unsuccessful per-VCI progress attempts to complete an operation. We demonstrate the benefit of our hybrid per-VCI optimization in Figure 6. Communication throughput is 6.97× lower without per-VCI progress (All w/o per-VCI progress) compared with the case where all optimizations are used.

Per-VCI request management. Even when operations from multiple threads map to different VCIs, they contend on the request-class's lock when they need to acquire a request (e.g., during an `MPI_Isend`) or release it (e.g., during an `MPI_Wait`). To address this contention, we maintain a cache of requests for each VCI. Access to each cache is protected by the VCI's lock. During the creation of a request, we first attempt to acquire a request from the cache belonging to the VCI that the operation maps to. This does not require acquiring an extra lock because the lock for the VCI is already held for the operation. If the cache is empty, we fall back on acquiring a request from the global pool, which requires acquiring the request class's lock. The caching idea extends to releasing a

---

§Features like OFI scalable endpoints can reduce the connection establishment and tear-down overheads, because they share the same transport-level address. However, we have not used them in this work because their performance is still not on par with that of regular endpoints, at least for the PSM2 provider that we used in this work. Furthermore, scalable endpoints share some resources, such as the OFI address vector, accesses to which could be serialized in the critical path by the OFI provider [10].

¶conceptually similar to an MPI_Send following the rendezvous protocol.

Table 1: Summary of locks on the critical path of initiation and progress operations in different critical sections.

Critical section \ MPI op.	Isend	Isend (immediate)	Put	Wait	Wait (immediate)
Global	1 (Global)	1 (Global)	1 (Global)	1 (Global)	1 (Global)
FG	2 (VCI + Request)	1 (VCI)	1 (VCI)	2 (VCI + Request)	0
FG + per-VCI req-cache	1 (VCI)	1 (VCI)	1 (VCI)	2 (VCI + VCI (request freeing))	0

request to the cache of a VCI as well. Thus, in the common case, we reduce the number of lock acquisitions in initiation operations to 1 (FG+per-VCI req-cache in Table 1, which summarizes the locks taken in different critical sections). Although the request class's lock is not taken (in the common case) for progress functions either, the VCI's lock pertaining to the request is taken twice—the final freeing of the request occurs in the MPI runtime layer, outside the critical section that protects the progress of the VCI.

In addition to traditional requests, MPICH maintains the pre-completed lightweight request described in Section 4.1. A light-weight request is a single object and not a pool, so it cannot be cached like traditional requests. What we do instead is replicate this lightweight request and provide each VCI with its own. The per-VCI lightweight requests do not need atomic operations for their updates since each is protected by the lock of the VCI it belongs to.

Figure 7 shows the benefits of the per-VCI request management optimizations. Without the optimizations, throughput is 39.98× lower (All w/o per-VCI req-mgmt) compared with all optimizations.

Cache-line awareness for VCIs. We implement the VCI pool as an array of structs. Each VCI struct holds the lock for that VCI. Locks of consecutive VCIs are likely to lie on the same cache line, resulting in the effects of false sharing when threads map to different VCIs. Hence, we use compiler attributes to cache-align each VCI. Figure 8 shows that without a cache-aware VCI, the message rate is 1.49× lower (All w/o cache-aware VCI).

Summary. All the thread-safety and multi-VCI optimizations described in this section are critical for enabling fast parallel streams of communication for MPI+threads. The message rate achieved by the optimized MPI library with 16 threads for 8-byte `MPI_Isends` is 94.43× higher than that of the state of the art.

5 MICROBENCHMARK ANALYSIS

We first measure the aggregate message rate of `MPI_Isend` and `MPI_Put` (passive target synchronization) using a communication-intensive benchmark. The benchmark demonstrates the maximum rate at which multiple cores can inject messages into the network simultaneously. Each core on the host node targets a distinct core on the remote node. We compare the following modes of execution.

- MPI everywhere parallelism using the original MPICH version that uses one VCI.
- MPI+threads (ser_comm+orig_mpich) parallelism with the user not exposing communication parallelism on the original MPICH that uses one VCI and the Global critical section.
- MPI+threads (ser_comm+vcis)—same as above but using the optimized multi-VCI based MPICH/CH4.
- MPI+threads (par_comm+orig_mpich) parallelism with user-exposed parallelism on the original MPICH.
- MPI+threads (par_comm+vcis)—same as above but using the optimized multi-VCI based MPICH/CH4.

- MPI+threads parallelism with user-visible endpoints on top of the optimized multi-VCI infrastructure where each endpoint is a VCI.

User-visible endpoints enable explicit control over VCIs, allowing users to specify the endpoint to use on the host and the remote endpoint to target. The communication performance of user-visible endpoints reflects the upper bound of the MPI-3.1 implementation wherein users implicitly use VCIs through MPI-3.1 mechanisms.

For our analysis with MPI+threads, we spawn one rank per node with an OpenMP thread per core. MPI everywhere uses a rank per core. In our microbenchmarks with user-visible endpoints, each host-target thread pair uses its own endpoint, thus exposing communication parallelism to the MPI library. With MPI-3.1, when users do not expose parallelism (ser_comm), all threads use the same communicator or window. In user-exposed parallelism (par_comm), each thread pair uses its own communicator or window.

5.1 Well-behaved communication

For the different modes of execution on OFI/OPA and UCX/IB, Figure 10 shows the message-rate scalability of a small-message `MPI_Isend`, and Figure 11 shows the message rate of `MPI_Isend` with 16 cores across varying message sizes. MPI everywhere achieves the highest throughput in all cases. When users expose communication parallelism, they achieve the same performance irrespective of the use of VCIs (par_comm) or user-visible endpoints. When users expose no communication parallelism (ser_comm), there is no performance gain with increasing number of threads.

Thread safety costs. A corresponding MPI everywhere configuration represents the practical upper bound of the communication performance of an MPI+threads configuration. Our optimized MPI+threads library utilizes the same level of network parallelism as MPI everywhere. However, MPI+threads incurs thread safety overheads over MPI everywhere even in the uncontended case. These overheads are most visible for small messages (see Figure 11) since the message rate is bound by the CPU, not by the network. The sources of the thread safety overheads are lock acquisitions and atomics for completion or reference counting. Figure 12 shows that if we disable locking and atomics,[||] MPI+threads can match the throughput of MPI everywhere. One solution to mitigate thread-safety costs would be to allow users to provide hints as to which communicators will be accessed by a dedicated thread, thereby allowing the MPI library to disable locking for the VCIs of communicators accessed by dedicated threads.

> *Takeaway:* For basic communication, VCIs and endpoints perform similarly and nearly as well as MPI everywhere.

[||]Since each thread maps to its own VCI in the MPI+threads microbenchmark, disabling thread safety, although incorrect, does not lead to erroneous behavior.
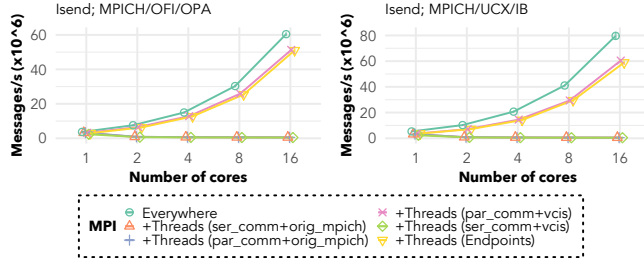
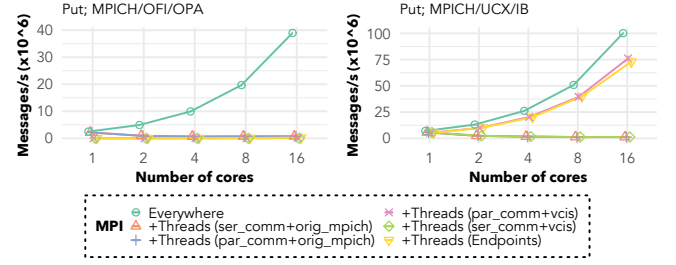Figure 10: Message-rate scalability of 8-byte `MPI_Isend`.

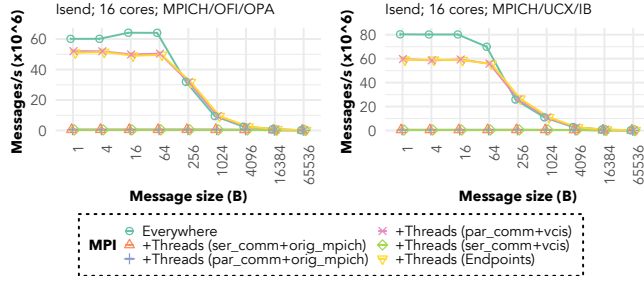Figure 13: Message-rate scalability of 8-byte `MPI_Put`.

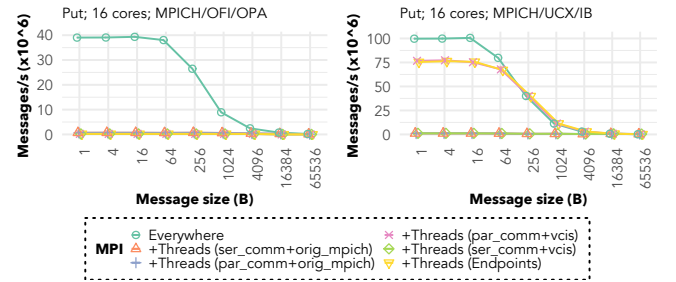Figure 11: `MPI_Isend` throughput with varying message sizes.

Figure 14: `MPI_Put` throughput with varying message sizes.
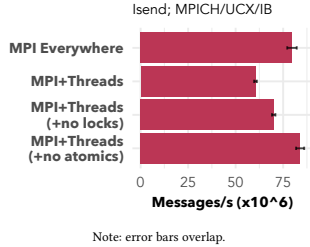

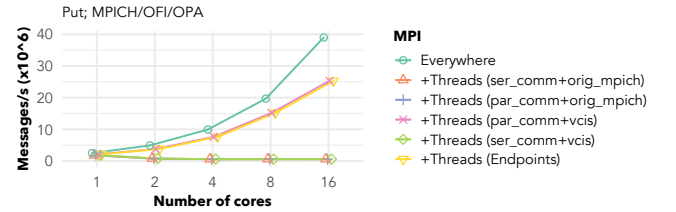Note: error bars overlap.

Figure 12: MPI+threads costs.

Figure 15: Parallel Win_free.

5.2 Not-so-well-behaved communication

Similar to Figures 10 and 11, Figures 13 and 14 demonstrate, for the different modes of execution on OFI/OPA and UCX/IB, the throughput scalability of a small-message `MPI_Put`, and the 16-core message rate of `MPI_Put` across varying message sizes, respectively

Network hardware limitations. The MPI+threads message rate of `MPI_Put` on OFI/OPA is dismal even with exposed parallelism on VCIs and user-visible endpoints. The reason is that Intel OPA emulates its RMA operations in software, requiring the application on the target side to actively progress a VCI for a performance-oriented execution of the operation. When the application provides no help, OPA relies on its low-frequency PSM2 progress thread for completion of the operation. In our benchmark, all the threads from all processes first initiate their RMA operations in parallel. Then, one thread waits on an MPI barrier, after which all threads synchronize with a thread barrier. The communicator used for the MPI barrier internally uses a VCI different from those of the windows on which the RMA operations are issued. Thus, none of the threads directly make progress on the incoming messages of the RMA VCIs.

The thread waiting on the MPI barrier occasionally performs global progress, so the benchmark eventually completes, but such global progress is infrequent and thus hurts performance.

With UCX/IB, on the other hand, we see no such degradation in performance because Mellanox IB is capable of implementing contiguous `MPI_Put` operations fully in hardware. Thus, even if the target threads are not making direct progress on the RMA VCIs, the operations still complete quickly.

With MPI everywhere, each process has a single VCI. Thus, the target ranks waiting on an MPI barrier continuously progress the VCI being targeted by the initiator ranks.

The main point demonstrated here is the tradeoff between dedicated progress and shared progress. MPI everywhere has no distinction between dedicated and shared progress because it only has a single VCI. For MPI+threads, when a single VCI is used (*i.e.*, original MPICH), like MPI everywhere, it has no distinction between dedicated and shared progress either. But, for MPI+threads, when we use multiple VCIs, the same independence of VCIs that enables good performance through the avoidance of locks also hurts shared
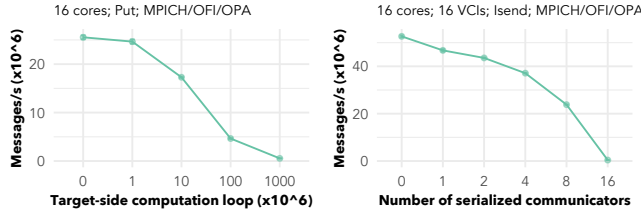
Figure 16: Busy target.   Figure 17: Mapping mismatch

Figure 18: Dedicated threads. Figure 19: Hurtful abstraction

progress between the threads. One can work around this issue by, for example, having each thread be responsible for progress on its window (in the same way that MPI everywhere works). One possibility is that threads call `MPI_Win_free` on their own windows in parallel (see Figure 15), thus making progress on the corresponding VCIs, although how practical this possibility is in real applications remains to be seen.

Busy target. Typically, the target side is involved in its own computational activities and does not just wait for communication to complete, as in Figure 15. The target's computation then determines the productivity of operations that need the target VCI to be progressed. Figure 16 shows a deteriorating `MPI_Put` message rate when the computation before the call to `MPI_Win_free` increases on the threads of the target rank.

> *Takeaway:* When shared progress is required neither VCIs nor user-visible endpoints perform well.

Mismatch in expected mapping to VCI. Even if the user exposes parallelism, parallel operations can contend on the same VCI because the number of VCIs available is hardware dependent and typically small, and these VCIs themselves are not exposed to the user. A simple first-come, first-served model of VCI allocation to communicators might not be the best strategy in this regard because the user cannot identify which communicators map to distinct VCIs and can therefore be used by different threads for better performance. Figure 17 shows the deteriorating effect on throughput for increasing amounts of serialization when there are 16 threads and the network hardware features only 16 contexts. The user is exposing communication parallelism, but the observed performance is low because of the mismatch in expectations of mapping to the underlying VCIs. One solution to this mismatch in expected mapping would be to allow users to provide hints as to which communicators are used by different threads and can benefit from independent VCIs.

User-visible endpoints, compared to MPI-3.1, can perform better in this situation because they expose the network hardware contexts to the user. We note that user-visible endpoints could also be implemented as a virtual layer on top of internal VCIs. However, they form a closer mapping to network resources than what communicators do.

> *Takeaway:* User-visible endpoints allow users to carefully manage their communication, thus performing better in situations when the MPI-3.1 library serializes user-exposed parallelism.
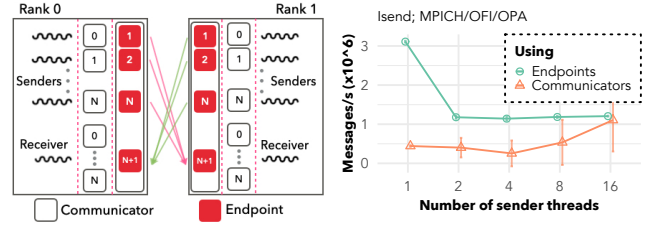
5.3 Limiting MPI semantics

Abstracting the use of VCIs through communicators can hurt certain irregular communication patterns. Figure 18 captures the communication pattern of Legion's [17] runtime, which maintains a set of threads where a few are dominant message senders and a few are dedicated polling threads that receive messages. With MPI-3.1, each sender thread uses a separate communicator. However, the semantics of MPI require the receiver thread to iterate over the communicators, thus forcing the receiver to contend on the VCIs of the senders and hurting performance (see Figure 19). With user-visible endpoints, this contention does not exist since each thread uses a distinct endpoint and can directly address the endpoint of the remote receiver. The single receiver is a bottleneck with both user-visible endpoints and communicators. With communicators, the fraction of time spent by the receiver on a VCI's lock decreases with increasing number of senders. Hence, its performance approaches that with endpoints.

> *Takeaway:* When MPI's semantics limit the user from exposing parallelism, user-visible endpoints perform better than VCIs.

6 APPLICATION ANALYSIS

In this section, we showcase the communication performance of three applications, one from each of the three categories described in Section 1. For each application, we compare the performances of MPI everywhere (a rank per core) and MPI+threads (a rank per node; an OpenMP thread per core) parallelism. For MPI+threads, we show the performances of user-exposed parallelism on VCIs and on the original MPI library and compare them with that of user-visible endpoints.

6.1 Stencil applications

Stencils are arguably the most common design patterns in HPC applications. They are at the heart of various application domains such as computational fluid dynamics, image processing, and partial differential equation solvers. Prominent applications with the stencil communication pattern include Nek5000 [33] and LAMMPS [34].

Using a 2D 5-point stencil, we evaluate the neighborhood halo exchange (non-blocking point-to-point) time per iteration of the stencil pattern. We first partition the mesh into blocks across nodes, and then within each node we further partition the sub-block among cores (Figure 20 shows an example). The squares formed by the intersection of the dashed blue lines represent cores that are driven
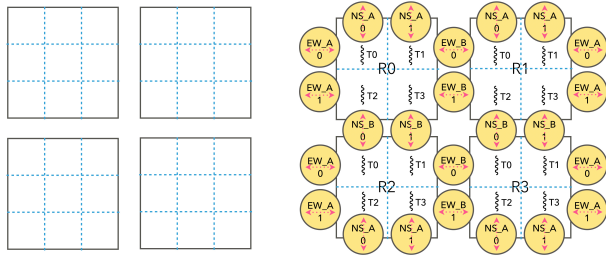
Figure 20: 6x6 grid with 3x3 sub-blocks per node.

Figure 21: Logical parallelism in MPI+threads stencil.

Figure 22: Halo communication across varying mesh sizes.

by processes and threads in MPI everywhere and MPI+threads parallelism, respectively. The blue dashed lines also represent boundaries where the halo exchange takes place through shared memory. MPI still executes intranode halo exchanges in MPI everywhere. In MPI+threads, threads use MPI only for internode halo exchanges and directly read the shared memory for intranode communication. The stencil pattern falls into the first category of applications—the internode communication of threads on edges of the nodes is independent and can execute on its own communication stream.

With user-visible endpoints, we create as many endpoints as there are threads on edges. For the example in Figure 20, we create 8 endpoints per node. Each communicating thread uses its own endpoint and exchanges halos by addressing the ranks of the remote endpoints, thereby achieving parallel communication. With MPI-3.1, we use two sets of communicators—odd and even—for each of the north-south and east-west exchanges. Each set contains as many communicators as there are threads on the node edge. Figure 21 shows an example. Depending on the Cartesian coordinates of the rank, the threads on a rank would use either the odd set or the even set. The odd-even sets prevent multiple threads from using the same communicator. Without them, T0 on R0 and R2 in Figure 21 would use the same NS_0 communicator, requiring T2 on R0 to also use the NS_0 communicator and thus serializing the communication of T0 and T2 on R0. Periodic stencils where the number of ranks along a dimension of the process-grid is odd require a separate set of communicators for the wraparound. The communicator usage can indeed be reduced without hurting performance by using only one communicator for the threads on corners, since their halo exchanges execute in serial.

The above example also demonstrates that the matching semantics of communicators or tags (i.e., the same communicator and tag must be used for both the sender and receiver) sometimes makes exposing communication parallelism with MPI-3.1 clumsy compared with that of user-visible endpoints. While not a performance argument, one might consider it to be a productivity concern.

Our evaluation utilizes all 9 nodes of the OFI/OPA cluster and engages 16 cores per node. Figure 22 shows the halo communication time** for each mode across varying mesh dimensions. This time discards the cost of any load imbalance since we use MPI barriers before the start of each halo exchange. We observe that

---

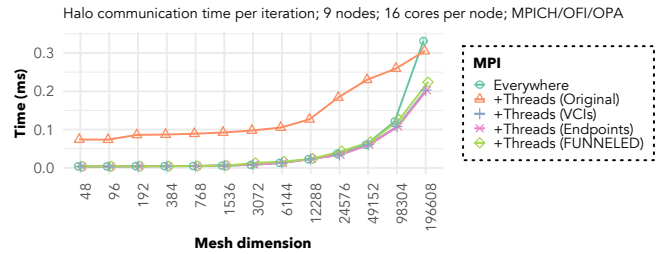**For the MPI_THREAD_FUNNELED mode, we do not report the time spent in packing and unpacking the buffer.

the communication performance of VCIs with user-exposed parallelism matches that of MPI everywhere parallelism, user-visible endpoints, and MPI_THREAD_FUNNELED.

Recommendation: Maximize independence between threads for point-to-point communication with MPI communicators.

Warning: Independent communication with MPI ranks or tags is not sufficient because of wildcards on the receive side.

Warning: The matching requirements of communicators or tags sometimes makes exposing communication parallelism with MPI-3.1 clumsy compared with that of user-visible endpoints.

6.2 OpenMC

The Center for Exascale Simulation of Advanced Reactors (CESAR) was a DOE co-design center whose primary objective was to adapt algorithms to the next-generation HPC architectures on the path to exascale systems. CESAR focused on algorithms that target the high-fidelity analysis of nuclear reactors. These include algorithms governing thermal hydraulics and neutronics. Applications simulating the former typically have a neighborhood, stencil style of communication, which we evaluated in Section 6.1. The latter consists of distributed Monte Carlo (MC) neutron-transport codes, such as OpenMC [37]. Siegel et al. [39] presented the original energy-banding (EB) algorithm for OpenMC, and Felker et al. [23] extended the EB idea to distributed-memory machines by distributing the cross-section data (composed of energy bands) across multiple nodes. Rather than the domain, particles are evenly distributed between the nodes. During simulation, each node fetches one band of the cross section using MPI_Get operations, tracks the movement of its share of particles, and iterates over the number of bands.

CESAR's EBMS miniapp [2] captures the communication pattern of the distributed EB idea. It utilizes MPI shared memory [28]: multiple processes on a node that share a receive buffer that is large enough to hold one band of the cross-section. While the computation is distributed among the different processes on the node, only one process is responsible for communication. We extended the EBMS miniapp to distribute the communication workload among the processes as well [3]. We also implemented a MPI+threads version of the miniapp with one multithreaded process per node. The communication workload between the cores is the same for
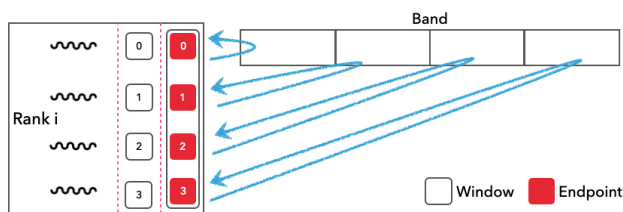
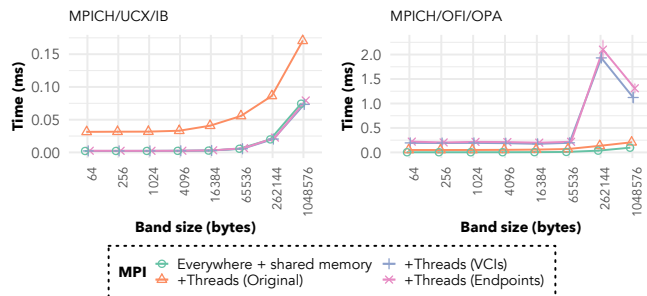**Figure 23: Logical parallelism in MPI+threads EBMS.**

Figure 24: Time per remote fetch across varying band sizes with 16 cores per node on UCX/IB (left) and OFI/OPA (right).
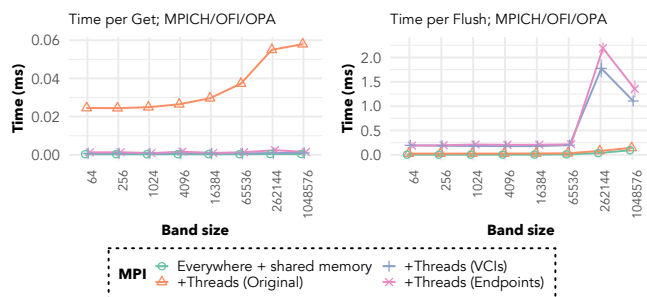

Figure 25: Get and flush time across varying band sizes on OFI/OPA.

both the MPI everywhere (+ shared memory) and the MPI+threads versions.

The EBMS pattern falls into both the first and second categories of applications (listed in Section 1). It falls into the first category because `MPI_Get` operations of different threads are independent; they can execute on distinct communication streams. The pattern falls into the second category because of the use of RMA—the underlying interconnect may be limited and rely on shared progress.

To leverage the independence between threads with user-visible endpoints, we create a separate endpoint for each thread. With MPI-3.1, we use a separate window per thread as shown in Figure 23. The memory is not duplicated for each window.

Our evaluation utilizes 4 nodes and engages 16 cores per node on both the UCX/IB and OFI/OPA clusters. We measure the time for each fetch of a portion of a band that resides on a remote node. A remote fetch includes an `MPI_Get` and an `MPI_Win_flush`. Figure 24
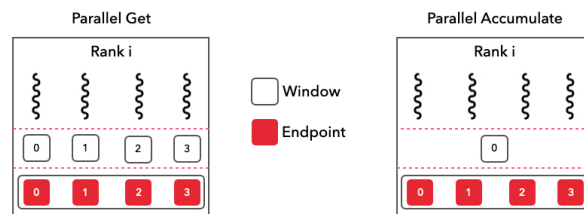

Figure 26: Logical parallelism in MPI+threads BSPMM.

shows the time for a remote fetch on the UCX/IB cluster. The communication performance of MPI+threads with VCIs is the same as that of MPI everywhere and user-visible endpoints.

> *Recommendation:* Maximize independence between threads for RMA communication with MPI windows.

On the other hand, the remote-fetch times on OFI/OPA (see Figure 24) show that exposing parallelism on VCIs hurts performance, especially for large messages. The case is the same with user-visible endpoints. The time for a remote fetch is governed by the issue of the fetch (`MPI_Get`) and its completion (`MPI_Win_flush`). If we separate them out, Figure 25 shows that the time for an `MPI_Get` using multiple VCIs is the same as that in MPI everywhere but the time of `MPI_Win_flush` is more expensive. The reason is that the communication pattern of the application does not guarantee that the remote VCI being targeted by the `MPI_Get` operations will be progressed—the thread mapped to the target VCI on the remote rank could be waiting on a thread-barrier that exists between each iteration of the simulation. Intel OPA relies on the application to make progress on the target VCI for the completion of large-message RMA transfers and for a productive execution of small to medium message transfers. Hence, the execution is dependent on the occasional global progress in the progress engine.

> *Warning:* Independent communication with VCIs fundamentally opposes shared progress.

6.3 NWChem

NWChem [41] is a prominent quantum chemistry application suite for large-scale simulations of chemical and biological systems. It uses the Global Arrays (GA) [31] library to distribute the multidimensional arrays across the memories of multiple nodes and provide access to the data through one-sided MPI operations. When NWChem is used for quantum chemical many-body methods, such as CCSD and CCSD(T), the dominant cost is that of BSPMM: block-sparse matrix multiplication (tensor contractions). NWChem implements this with dense matrix operations using a *get-compute-update* pattern: each worker (processing entity) uses `MPI_Get` to retrieve the submatrices it needs, and after the multiplication it uses an `MPI_Accumulate` to update the memory at the target location.

Using a mini-app [1], we evaluate a 2D version of this communication pattern that performs $A \times B = C$, wherein the input matrices A and B are composed of tiles. Each tile is either a dense or zero matrix. The nonzero tiles are evenly distributed among the ranks
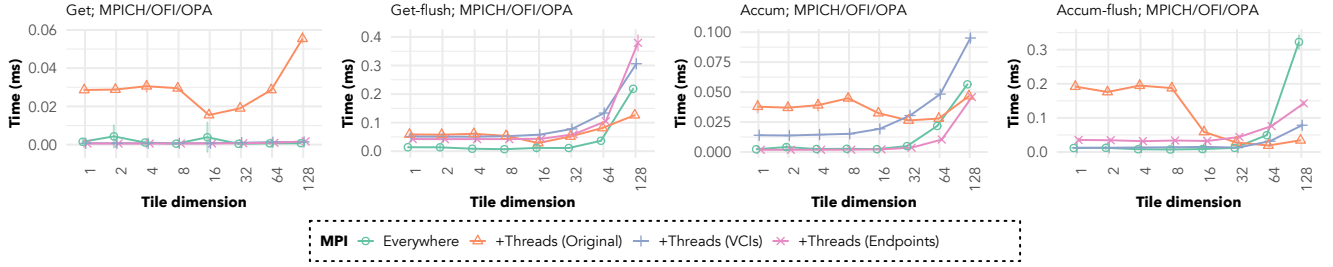
Figure 27: BSPMM communication performance on Intel Omni-Path.

in a round-robin fashion. Each rank maintains a work-unit table that lists all the multiplication operations that workers need in order to cooperatively execute. Rank 0 hosts a global counter, which the workers fetch and add atomically (`MPI_Fetch_and_op`). The fetched counter serves as an index to the work-unit table. Each worker locally accumulates its C tiles until the next fetched work unit corresponds to a different C tile, in which case the worker uses an `MPI_Accumulate` to update the C tile. A worker is a process in MPI everywhere and a thread in MPI+threads.

MPI+threads BSPMM falls under the third category of applications. Although each thread can use its own window for its `MPI_Get` to fetch tiles of A and B, MPI-3.1's semantics constrain the threads within a rank to use a single window for the `MPI_Accumulate`. Each thread cannot use its own window for `MPI_Accumulate` because atomicity across windows for the same memory location is undefined. On the other hand, user-visible endpoints enable the creation of multiple endpoints within a single window. Hence, each thread uses its own endpoint for both `MPI_Get` and `MPI_Accumulate` as shown in Figure 26.

Figure 27 portrays the performance of BSPMM's communication pattern on 4 nodes of the OFI/OPA cluster with 16 cores engaged per node. We measure the time taken to initiate the operations (e.g., `MPI_Get`) separately from the time taken to complete them (e.g., `MPI_Win_flush`). VCIs initiate `MPI_Get` operations as fast as endpoints and MPI everywhere. However, only endpoints initiate `MPI_Accumulate` operations as fast as MPI everywhere; MPI+threads with MPI-3.1 is constrained by the use of a single window. The flush of `MPI_Get` operations demonstrates behavior similar to that in the EBMS pattern (see Section 6.2). MPI+threads with VCIs flushes `MPI_Accumulate` operations faster than endpoints because of its use of a single VCI—the probability of the remote target VCI being progressed is higher since all threads on the target rank map to it. The Accum-flush of MPI everywhere is the slowest for large tile dimensions because the worker cannot progress its VCI until it finishes its computational tasks, which is larger for large tile dimensions. On the other hand, if a worker in MPI+threads is busy with computational tasks, other workers on the same rank might progress the VCI, either because they all map to the same VCI or because of shared progress, allowing for a more productive execution of a large-message RMA operation than that in MPI everywhere.

> *Warning:* Atomic operation semantics are not easy to achieve with multiple windows; using multiple VCIs may not help.

An important point to note is that the `MPI_Accumulate` operations in BSPMM do not need to be ordered. Hence, if the user hints this relaxation using the `accumulate_ordering=none` hint, the MPI library could issue the operations from different threads in parallel and thereby achieve the same performance as user-visible endpoints. Furthermore, increasing the number of ranks per node and decreasing the number of threads per rank in hybrid MPI+threads could also help the case of accumulates with VCIs. The optimal combination of ranks per node and threads per rank, however, depends on an empirical study with the application.

7 RELEVANCE TO MPI-4.0

The next iteration of the MPI standard, MPI-4.0 [7], is considering featuring new info hints (e.g. `mpi_assert_no_any_tag`) that would provide the users with more opportunities to expose communication parallelism in their MPI+threads communication. For example, if an application hints that it does not use wildcards in its communication, MPI-4.0 would allow the user to expose communication parallelism through tags within a single communicator in addition to the option of exposing parallelism through communicators. These new ways to expose parallelism would, in turn, need to be mapped to the multiple VCIs inside the MPI library. Hence, the productive use of the new hints relies on the muti-VCI infrastructure that this work provides.

8 RELATED WORK

The communication performance of MPI+threads has been a decade-long concern. Researchers have studied the problem in various ways, ranging from mitigating lock contention on the MPI library's software resources [12, 13, 16] to extending the MPI standard [20, 25]. We discuss prior works that are conceptually related to ours.

8.1 MPI Endpoints demonstration

Dinan et al. [22] and Sridharan et al. [40] demonstrate the performance of MPI+threads with the MPI Endpoints proposal. Although they agree that using a separate communicator per thread would allow the user to expose parallelism, they do not compare MPI Endpoints with the communicator-based approach. Our work compares the capabilities of the existing MPI standard with user-visible endpoints, demonstrating scenarios where VCIs do as well as endpoints and where they falter. Additionally, their work does not describe the notion of progress, which is critical for correctness. Our work, on the other hand, does not sacrifice correctness for performance.

8.2 MPI libraries

Open MPI. A couple of works [24, 32] on Open MPI are conceptually similar to our work—they use fine-grained critical sections and map parallelism available in the existing MPI standard to multiple network hardware contexts to improve MPI+threads communication performance. However, both works do not compare against user-visible endpoints or MPI everywhere. Additionally, like the MPI Endpoints work, neither of these works discusses the notion of shared progress, ignoring correctness.

Gopalkrishnan et al. [24] evaluate the communication performance of MPI+threads with OFI scalable endpoints. Recognizing the practical performance limitations of scalable endpoints, our work uses regular OFI endpoints instead (see Section 4.2), and hence we observe much larger speedups than their work obtains.

Similar to VCIs in our work, Patinyasakdikul et al. [32] define Communication Resource Instances (CRIs). Their approach involves creating a pool of CRIs and either assigning CRIs to operations in a round-robin fashion or assigning CRIs to threads using thread-local storage. While this approach may be correct for a subset of operations, some CRIs break MPI's semantics for operations such as `MPI_Accumulate` operations to the same target location. Such operations are ordered by default on a window. In terms of performance, even with user-exposed parallelism their point-to-point communication performance does not scale with increasing number of threads unlike the results of our work.

Intel MPI. Since its 2019 release, the Intel MPI library has utilized multiple network hardware contexts on Intel Omni-Path through its multiple endpoints support [5]. However, this support is only for a nonstandard threading level: MPI_THREAD_SPLIT, which does not cover all cases possible in the MPI_THREAD_MULTIPLE threading level. In contrast, our work with VCIs fully and correctly supports MPI_THREAD_MULTIPLE.

In this work, we do not compare against the capabilities of other MPI libraries since our goal is not to show that we can do better than they can; rather, our aim is to study the strengths and limitations of MPI-3.1 compared with those of user-visible endpoints. Adding other libraries into the mix would blur the analysis in this paper. However, a separate performance comparison with other libraries would make a worthy future study.

8.3 Distributed-memory programming models

The newest version of the OpenSHMEM specification features user-visible network contexts. Dinan et al. [21] evaluate this approach. Although our work is not on OpenSHMEM, the motivation to improve multithreaded communication is the same. Instead of leaping into extending the standard, however, we evaluate the capabilities of the existing MPI-3.1 standard for MPI+threads. Given the strengths and limitations of MPI showcased in this paper, the MPI community is better equipped to propose extensions to MPI, if any.

9 CONCLUDING REMARKS

The MPI+threads programming model is critical for effectively utilizing modern processors. To dissolve its communication bottleneck, however, domain scientists must expose logical parallelism in their communication. Only then will we able to achieve the true potential of MPI+threads. The school of thought so far has been

that we need user-visible endpoints to express logical parallelism. In this paper, however, we show that the existing MPI standard already allows its users to overcome its ordering constraints and express parallelism. By mapping MPI-3.1's parallelism to internal virtual communication interfaces, in the majority of cases we can achieve communication performance equal to the performance of user-visible endpoints and MPI everywhere without sacrificing correctness. More important, domain scientists do not need to worry about managing and mapping to the limited hardware resources with MPI 3.1, which is not the case in a user-visible solution such as MPI Endpoints. We expect that MPI-4.0 will increase the opportunities for users to express parallelism through hints, and that, with the adoption of VCIs, MPI developers will be able to effectively exploit new and current ways of expressing logical communication parallelism.

REFERENCES

[1] [n.d.]. BSPMM mini-app. https://github.com/rzambre/bspmm.
[2] [n.d.]. EBMS mini-app. https://github.com/ANL-CESAR/EBMS.
[3] [n.d.]. Extended EBMS mini-app. https://github.com/rzambre/ebms.
[4] [n.d.]. Intel Omni-Path Fabric Host Software. https://www.intel.com/content/dam/support/us/en/documents/network-and-i-o/fabric-products/Intel_OP_Fabric_Host_Software_UG_H76470_v9_0.pdf.
[5] [n.d.]. Intel® MPI Multiple Endpoints Support. https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support.
[6] [n.d.]. Mellanox PRM. http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.
[7] [n.d.]. MPI-4.0 Draft Report. https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf.
[8] [n.d.]. MPI Endpoints. https://github.com/mpi-forum/mpi-issues/issues/56.
[9] [n.d.]. Semantics of Point-to-Point Communication. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node58.htm.
[10] [n.d.]. Shared AV table in OFI/PSM2. https://github.com/ofiwg/libfabric/issues/5080.
[11] [n.d.]. TOP500 Meanderings: InfiniBand Fends Off Supercomputing Challengers. https://www.top500.org/news/top500-meanderings-infiniband-fends-off-supercomputing-challengers/.
[12] Abdelhalim Amer, Charles Archer, Michael Blocksome, Chongxiao Cao, Michael Chuvelev, Hajime Fujita, Maria Garzaran, Yanfei Guo, Jeff R Hammond, Shintaro Iwasaki, et al. 2019. Software combining to mitigate multithreaded MPI contention. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, 367–379.
[13] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+ threads: Runtime contention and remedies. *ACM SIGPLAN Notices* 50, 8 (2015), 239–248.
[14] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. 2010. PMI: A scalable parallel process-management interface for extreme-scale systems. In *European MPI Users' Group Meeting*. Springer, 31–41.
[15] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2008. Toward efficient support for multithreaded MPI communication. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 120–129.
[16] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2010. Fine-grained multithreading support for hybrid threaded MPI programming. *The International Journal of High Performance Computing Applications* 24, 1 (2010), 49–57.
[17] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of*

the International Conference on High Performance Computing, Networking, Storage
and Analysis. IEEE, 1–11.

[18] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata,
Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Ge-
offroy R Vallee. 2017. A survey of MPI usage in the U.S. Exascale Computing
Project. Concurrency and Computation: Practice and Experience (2017), e4851.

[19] Aydin Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patter-
son. 2017. Distributed-memory breadth-first search on massive graphs. arXiv
preprint arXiv:1705.04590 (2017).

[20] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev
Thakur. 2013. Enabling MPI interoperability through flexible communication
endpoints. In Proceedings of the 20th European MPI Users' Group Meeting. ACM,
13–18.

[21] James Dinan and Mario Flajslik. 2014. Contexts: a mechanism for high throughput
communication in OpenSHMEM. In Proceedings of the 8th International Conference
on Partitioned Global Address Space Programming Models. ACM, 10.

[22] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc
Snir, and Rajeev Thakur. 2014. Enabling communication concurrency through
flexible MPI endpoints. The International Journal of HPC Applications 28, 4 (2014),
390–405.

[23] Kyle G Felker, Andrew R Siegel, Kord S Smith, Paul K Romano, and Benoit
Forget. 2014. The energy band memory server algorithm for parallel Monte
Carlo transport calculations. In SNA+ MC 2013-Joint International Conference on
Supercomputing in Nuclear Applications+ Monte Carlo. EDP Sciences, 04207.

[24] Aravind Gopalakrishnan, Matias A Cabral, James P Erwin, and Ravindra Babu
Ganapathi. 2019. Improved MPI Multi-Threaded Performance using OFI Scalable
Endpoints. In 2019 IEEE Symposium on High-Performance Interconnects (HOTI).
IEEE, 36–39.

[25] Ryan E Grant, Matthew GF Dosanjh, Michael J Levenhagen, Ron Brightwell, and
Anthony Skjellum. 2019. Finepoints: Partitioned multithreaded mpi communi-
cation. In International Conference on High Performance Computing. Springer,
330–350.

[26] 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.

[27] Edward Higgins, Matt Probert, Phil Hasnip, Keith Refson, and Ian Bush. 2015.
Hybrid OpenMP and MPI within the CASTEP code. Technical Report. ARCHER
eCSE Technical Report.

[28] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron
Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2013. MPI+ MPI: A
new hybrid approach to parallel programming with MPI plus shared memory.
Computing 95, 12 (2013), 1121–1136.

[29] Daniel Holmes. [n. d.]. Introducing Endpoints into the EMPI4Re MPI library.
([n. d.]).

[30] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and
Barbara Chapman. 2011. High performance computing using MPI and OpenMP
on multi-core parallel systems. Parallel Comput. 37, 9 (2011), 562–575.

[31] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1994. Global
arrays: a portable shared-memory programming model for distributed memory
computers. In Proceedings of the 1994 ACM/IEEE conference on Supercomputing.
IEEE Computer Society Press, 340–349.

[32] 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.

[33] James W. Lottes Paul F. Fischer and Stefan G. Kerkemeier. 2008. nek5000 Web
page. http://nek5000.mcs.anl.gov.

[34] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics.
J. Comput. Phys. 117, 1 (1995), 1–19.

[35] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. Hybrid MPI/OpenMP
parallel programming on clusters of multi-core SMP nodes. In 2009 17th Euromicro
International Conference on Parallel, Distributed and Network-based Processing.
IEEE, 427–436.

[36] Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland,
Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Block-
some, et al. 2017. Why is MPI so slow?: Analyzing the fundamental limits in
implementing MPI-3.1. In Proceedings of the International Conference for High
Performance Computing, Networking, Storage and Analysis. ACM, 62.

[37] Paul K Romano, Nicholas E Horelik, Bryan R Herman, Adam G Nelson, Benoit
Forget, and Kord Smith. 2014. OpenMC: A state-of-the-art Monte Carlo code for
research and development. In SNA+ MC 2013-Joint International Conference on
Supercomputing in Nuclear Applications+ Monte Carlo. EDP Sciences, 06016.

[38] Pavel Shamis et al. 2015. UCX: an open source framework for HPC network
APIs and beyond. In 2015 IEEE 23rd Annual Symposium on High-Performane
Interconnects. IEEE, 40–43.

[39] A Siegel, Kord Smith, K Felker, P Romano, Benoit Forget, and P Beckman. 2014.
Improved cache performance in Monte Carlo transport calculations using energy

banding. Computer Physics Communications 185, 4 (2014), 1195–1199.

[40] Srinivas Sridharan, James Dinan, and Dhiraj D Kalamkar. 2014. Enabling efficient
multithreaded MPI communication through a library-based implementation of
MPI endpoints. In Proceedings of the International Conference for High Performance
Computing, Networking, Storage and Analysis. IEEE Press, 487–498.

[41] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma,
Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L
Windus, et al. 2010. NWChem: A comprehensive and scalable open-source
solution for large scale molecular simulations. Computer Physics Comm. 181, 9
(2010), 1477–1489.

[42] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. 2018. Scalable
communication endpoints for MPI+ Threads applications. In 2018 IEEE 24th
International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 803–
812.