Lessons Learned on MPI+Threads Communication

Rohit Zambre, Aparna Chandramowlishwaran
Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA, USA
{rzambre,amowli}@uci.edu

***Abstract*—Hybrid MPI+threads programming is gaining prominence, but, in practice, applications perform slower with it compared to the MPI everywhere model. The most critical challenge to the parallel efficiency of MPI+threads applications is slow `MPI_THREAD_MULTIPLE` performance. MPI libraries have recently made significant strides on this front, but to exploit their capabilities, users must expose the communication parallelism in their MPI+threads applications. Recent studies show that MPI 4.0 provides users with new performance-oriented options to do so, but our evaluation of these new mechanisms shows that they pose several challenges. An alternative design is MPI Endpoints. In this paper, we present a comparison of the different designs from the perspective of MPI's end-users: domain scientists and application developers. We evaluate the mechanisms on metrics beyond performance such as usability, scope, and portability. Based on the lessons learned, we make a case for a future direction.**

***Index Terms*—Exascale computing, Message-oriented middleware, MPI Endpoints, MPI+OpenMP, MPI+threads, MPI_THREAD_MULTIPLE, Partitioned communication**

I. Introduction

The hybrid MPI+threads model is gaining prominence over the traditional MPI everywhere approach following the evolution of modern computing architectures. Over the last decade, the number of cores on a processor has grown disproportionately to the growth in other on-node resources such as memory, TLB space, and network resources (work queues and doorbell registers) [2], [59]. Consequently, domain scientists have witnessed their applications run out of memory with the memory-hungry MPI everywhere model on large problem sizes [21], [48], [51]. With MPI+threads (*e.g.*, MPI+OpenMP), on the other hand, applications are able to scale to much larger problems since the model enables users to utilize the many cores on a processor with threads *and* efficiently share the limited on-node resources between cores with a single process per node (or NUMA domain). For these reasons, modern event-based frameworks (*e.g.*, Legion [18] and YGM [50]) have not been developed with MPI everywhere from the start.

In terms of performance, however, MPI+threads applications tend to perform slower than their MPI everywhere counterparts in practice [21], [38], [44], [48]. The reason is that MPI+threads programming raises many new challenges, such as mitigating thread-synchronization overheads [37], [43], [53], and preventing performance-degrading memory accesses (*e.g.*, false sharing), that are not present in MPI everywhere. The most critical challenge, however, is the dismal communication performance of MPI+threads applications [13], [14], [16], [59], [68]. This challenge is a pressing bottleneck because most scientific simulation campaigns run close to the strong-scaling limit where communication has been demonstrated to occupy a significant portion of an application's runtime [52], [54], [62], [63].

Furthermore, as an application approaches the strong-scaling limit, the size of each individual message decreases, and the communication performance is limited by the rate of issuing messages rather than the network bandwidth. In this paradigm, the interoperability of threads with MPI is critical for the performance of an MPI+threads application.

MPI defines multiple levels of threading support going from a highly restrictive level—only one thread will execute—to a completely flexible level: multiple threads can execute MPI operations in parallel (*i.e.*, `MPI_THREAD_MULTIPLE`). According to a 2017 survey of applications chosen as candidates for the upcoming exascale systems and other application development reports, domain scientists prefer using the flexible `MPI_THREAD_MULTIPLE` level but do not do so currently primarily because of poor performance [12], [19], [24].

Recently (starting in 2019), however, MPI libraries have made significant strides in achieving scalable multithreaded communication performance that matches that of MPI everywhere. Figure 1(a) shows MPICH's support for high-speed `MPI_THREAD_MULTIPLE` performance in its latest 4.0 release as an example. The primary factor for this improved performance is the MPI library's ability to map *logically parallel communication*—operations that are not ordered according to MPI's semantics—to the underlying network parallelism [4], [49], [66], [67]. With such new capabilities, applications are able to achieve the best of both worlds—high scalability and high performance—with MPI+threads compared to MPI everywhere. Figures 1(b) and 1(c) demonstrate the performance impact of using logically parallel communication for traditional stencil-style workloads (Uintah computational framework [54], [68]) and modern data-centric workloads (Legion-based Circuit simulation [18], [68]).

In any application, the key to achieving fast MPI+threads communication is logically parallel communication. Without it, the new MPI libraries are ineffective. So, how can domain scientists and application developers expose such communication parallelism?

In this regard, the MPI community first pursued user-visible extensions to the standard in the form of MPI Endpoints [26], [28], [31]. An endpoint represents a logically independent

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

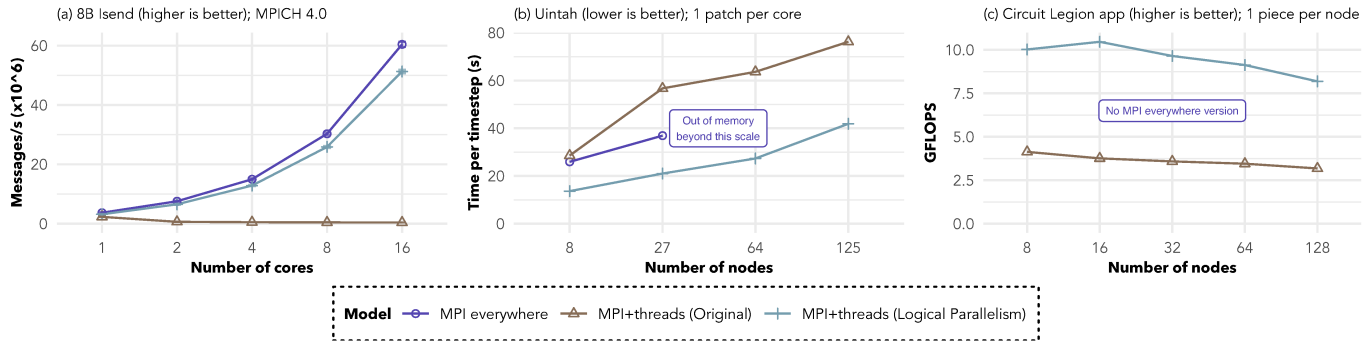

Fig. 1: MPI+threads achieves high scalability *and* high performance with logically parallel communication on new MPI libraries. All MPI+threads versions use `MPI_THREAD_MULTIPLE`; (a) uses Intel Skylake nodes, (b) uses Intel KNL nodes, and (c) uses Intel Broadwell nodes; the network in all experiments is Omni-Path.

stream of communication. Each endpoint is directly addressable through an MPI rank, making user-visible endpoints a flexible solution. The MPI Forum had deliberated the MPI Endpoints proposal but ultimately suspended it on the prospect that existing MPI objects such as communicators and windows can expose the same level of logical communication parallelism as user-visible endpoints. In cases where MPI's semantics prevent users from exposing communication independence, this school of thought advocates the use of MPI Info hints to relax the limiting MPI semantics which would allow the application to expose logically parallel communication through alternative MPI mechanisms like tags. To analyze the performance differences between the two approaches, recent studies map the capabilities of both to parallel network resources and evaluate them on applications from different domains: linear system solvers, graph analytics, astrophysics, particle physics, and event-based runtimes. These studies demonstrate that existing MPI mechanisms indeed perform as well as user-visible endpoints [67], [68], and they are the basis for the introduction of new hints for application-specific relaxation of semantics in the latest MPI 4.0 standard.

In this paper, we show that exposing the communication independence between threads with existing MPI objects has various drawbacks even with the new Info hints. Using communicators, the most explicit existing mechanism, to expose communication independence is quite complex even for applications with well-structured and regular communication patterns (*e.g.*, stencil workloads). The complexity arises from the fact that users need to ensure messages are matched appropriately *while* expressing communication parallelism. This complexity hurts not just the productivity of the domain scientist but also the MPI library's mapping to the underlying network parallelism, thereby hurting performance. Relaxing the unneeded MPI semantics through Info hints and expressing communication parallelism through tags instead is one way to combat this complexity, but such a solution is not portable since the optimal mapping to the parallel network resources is dependent on hints specific to an MPI library.

On the other hand, for the same applications, we find that the user-visible endpoints solution is not only portable but also straightforward because of the flexibility of its design: each endpoint is directly addressable. Like the idea of "ports" first described by Foster et al. [26], [31], endpoints do not constrain communication parallelism information with existing MPI semantics. The concern here, however, is the introduction of a new concept to the user. Presumably, this concern does not hold with existing MPI mechanisms since users are already aware of communicators, windows, etc. However, we argue that MPI users do not intuitively think of existing MPI objects as means of expressing parallelism. For example, users have historically viewed a communicator as a group of processes. That it can double up as a means to express logically parallel communication is a corollary of its definition. Hence, the concern of enforcing new concepts (*i.e.*, repurposing existing MPI objects) on users also holds for existing MPI mechanisms.

Finally, the MPI 4.0 standard also introduced partitioned operations as an alternative to MPI Endpoints to alleviate the performance problem of MPI+threads communication. This new interface allows users to define a persistent message with multiple data partitions so that each thread can drive different partitions of a single message. Ongoing research on this new interface demonstrates that partitioned communication does perform better than MPI+threads communication with no logically parallel communication ("MPI+threads (Original)" in Figure 1) [17], [30], [34], [65], but a performance analysis against the other two mechanisms, especially in its capabilities to utilize the underlying network parallelism, remains to be seen. We note, however, that the partitioned interface exhibits certain semantic limitations which could prevent it from matching the performance of the other approaches. For example, threads share the single MPI request of a partitioned message (see Section II-C) which means that threads would contend to access the shared request's resources in the MPI library, or that threads would need to synchronize to allow only one thread to poll for completion. In either case, the application is prone to incur overheads of contention or synchronization compared to fully independent paths. Techniques like double or triple buffering could partially mitigate the slowdowns caused by this limitation, but they do not allow threads to achieve complete independence in a manner that

the other mechanisms allow and promote.

Out of the three approaches, two have been introduced in MPI 4.0 and at least one of them allows users to expose complete independence between threads. So, does this mean *MPI's end users* are satisfied? As is the case with the success of any technology, the answer lies with the end users, the domain scientists. We show that MPI 4.0 does *not* meet the needs of MPI+threads applications, and that it introduces new problems. The current solutions in MPI 4.0 may be stepping stones to alleviating the performance of MPI+threads communication, but they are not sufficient by themselves.

In this paper, we compare the strengths and limitations of the designs (described in Section II) with respect to their applicability to MPI's point-to-point, RMA, and collective communication. Our comparison (see Section III) is centered around the lessons learned from evaluating a variety of different types of MPI+threads applications. Guided by both quantitative (in [67], [68]) and qualitative (this paper) comparisons of the different interfaces, in Section IV, we discuss a future direction that would enable applications to achieve not just high scalability and performance, but also high productivity with the MPI+threads programming model.

II. The Three Designs for MPI+Threads Communication

In this section, we describe how the three designs can expose communication independence between threads, and discuss their motivations and their implementation evaluations.

A. Existing MPI mechanisms

With existing MPI objects, such as communicators, tags, and windows, users can expose relatively unordered operations for all types: point-to-point, RMA, and collectives.

Communicators apply to collective and point-to-point communication. For the former, MPI requires collectives to be issued serially on a communicator. Hence, users can issue collectives in parallel on a process only on distinct communicators. For two-sided operations, MPI specifies a nonovertaking order and uses the ⟨communicator, rank, tag⟩ triplet to match operations. Two or more operations issued using different communicators cannot match the same target operation and hence imply no relative ordering; such operations are logically parallel. Operations that share a communicator but use different ranks or tags, however, are not logically parallel because of the possibility of wildcards (*e.g.*, `MPI_ANY_TAG`) on the receive side. Hence, the only way to expose logical communication independence for point-to-point operations with MPI's default semantics is through the use of multiple communicators.

Tags with hints. MPI 4.0 features new Info hints that allow an application to relax semantics it does not need. Info hints relevant to this paper include `mpi_assert_allow_overtaking`, `mpi_assert_no_any_tag`, and `mpi_assert_no_any_source`. The first, when set, informs the MPI library that the operations do not need to be matched in the order that they were posted. This Info hint is beneficial when the application requires wildcards but does not require the MPI library to maintain the order of matching. With the nonovertaking order relaxed, two or more send operations using different tags are logically parallel even if they use the same communicator and address the same target process. Because of wildcards, however, similar receive operations are not logically parallel. If the application does not require wildcards, the domain scientist can set the other two hints to relax the wildcard's constraints. Without any wildcards, two or more operations (both send and receive) that use the same communicator, address the same process, but use different tags can never match with the same target operation. Hence, such operations are logically parallel.

Windows apply to MPI's RMA operations. By default, MPI maintains program order only for its atomic operations (*e.g.*, `MPI_Accumulate`) originating from the same source and targeting the same memory location on the same window. Otherwise, both atomic and nonatomic RMA operations (*e.g.*, `MPI_Put`) on different windows are unordered. Although nonatomic operations are logically parallel in any case, users need to be wary of mixing synchronization and initiation operations in parallel on the same window. For example, if one thread is waiting inside `MPI_Win_flush` and another continuously issues `MPI_Get` operations, the first thread might block indefinitely. To overcome such cases and explicitly expose parallelism for any type of RMA operation, users have the option of using distinct windows for different threads.

By mapping the communication independence exposed by the above mechanisms to the underlying network parallelism, recent research demonstrates that MPI+threads applications can indeed achieve scaling communication throughput that matches that of MPI everywhere [49], [67], [68]

B. User-visible Endpoints

Researchers had initially proposed to extend the MPI standard to introduce user-visible MPI endpoints [28]. With a new API, users can create communicators with multiple endpoints (see Figure 2). This API creates a new communicator (context ID) from an existing one, `parent_comm`, and provides `my_num_ep` number of handles to the new communicator. Each handle is addressable with a distinct rank. For all operations, users would specify the local endpoint to use using one of the returned handles (`new_comm_handles`) and address a target endpoint using the endpoint's rank, a global index of the endpoint, making endpoints a flexible interface. One could then use an endpoints communicator to create endpoints for other MPI objects such as windows and files [5].

Each endpoint takes on the semantics of an MPI rank. Like messages originating from different processes, messages from different endpoints are unordered and hence logically parallel. If the user maps each thread to a distinct endpoint, then all

```
MPI_Comm_create_endpoints(parent_comm, my_num_ep, info,
    new_comm_handles)
```

Fig. 2: API to create a communicator with multiple endpoints.

threads are directly addressable. Given their flexible interface, user-visible endpoints represent the upper bound in expressing the communication parallelism available in an application. Several efforts show scaling multithreaded communication throughput with user-visible endpoints [29], [40], [58].

One of the notions on which the endpoints proposal was suspended was that some networking hardware may not be able to optimize the creation of new network addresses (for new endpoints) after initializing the MPI library [29]. We note, however, that the new MPI libraries have addressed this problem by creating a pool of network resources during the initialization phase [49], [67]. Such implementations then map logical entities like endpoints to physical network resources.

C. Partitioned Communication

MPI 4.0's partitioned communication interface allows users to specify a persistent message with multiple partitions. Each partition contributes to a single message, and the contributions could occur in parallel from multiple threads. Users define the operation's characteristics (*e.g.*, number of partitions, tag, etc.) outside the critical path, and then contribute the individual partitions of the message whenever a thread is ready to communicate (see Figure 3). MPI 4.0 contains APIs to describe standard-mode send and receive operations only, but the idea can extend to other modes of point-to-point operations and even RMA and collective operations [41]. These extensions will require their own set of APIs to define the partitioned communication equivalent for each operation.

```
MPI_Psend_init(buf, num_partitions, count, datatype,
               dest, tag, comm, info, request)
MPI_Precv_init(buf, num_partitions, count, datatype,
               source, tag, comm, info, request)
MPI_Pready(partition, request)
MPI_Parrived(partition, request, flag)
```

Fig. 3: New APIs to create and use standard mode send and receive partitioned communication operations.

Partitioned communication was introduced to combat the message-matching overheads in multithreaded communication. Message matching is a costly serial operation [56]. If n threads use the same communicator ("MPI+threads (Original)" in Figure 1), the overhead of message matching grows by $O(n)$. Since partitioned operations share a persistent message, they incur a message matching overhead of only $O(1)$ for n threads driving the multiple partitions of the message. Research implementations have demonstrated performance benefits of partitioned operations especially for large partitions even with older MPI libraries that do not capitalize on parallel network resources [30], [34]. Partitioned operations are suited to benefit from the capabilities of the new MPI libraries (multiple partitions could map to distinct network resources), but such a study has not been conducted yet. More important, how partitioned operations compare to the other mechanisms of exposing logically parallel communication where message matching is not a concern (due to a distinct matching engine per communication channel [49], [67]) remains to be seen.

III. Application-Centric Comparisons of The Three Designs

In this section, we discuss how the different designs (see Section II) compare against each other with respect to two key metrics: ease of use (which reflects the productivity of domain scientists), and applicability to different MPI operations (which measures the scope of the designs). We map the communication patterns of key applications to the different design choices to make such a comparison. In the process of doing so, we collaborate with application developers from a variety of institutions including University of Utah (stencil communication in the hypre linear solver used by Uintah [54]), Maison de la Simulation (stencil communication in Smilei [27]), Pacific Northwest National Laboratory (graph communication in Vite [33]), University of California, Irvine (stencil communication in Pencil [63]), HPE (RMA communication in WOMBAT [47]), and Argonne National Laboratory (Legion's MPI backend [6]). We organize our discussion below by the lessons learned from comparing the different designs.

A. Point-to-point communication

Mechanism 1: Communicators. Using distinct communicators is the most explicit way to express logically parallel communication for point-to-point operations with MPI's default semantics (see Section II-A). We discover, however, that communicators pose several challenges: complexity, high resource usage, and lack for flexibility for irregular and dynamic communication patterns. We detail our lessons learned below.

> *Lesson 1:* Exposing logically parallel communication with communicators is a complex task due to its matching requirements.

To understand the first lesson, let us consider a relatively simple example of a static (communication pattern of each thread is fixed) 2D 9-point stencil. Figure 4 shows the ideal communicator usage—minimum number of communicators with all of the available parallelism exposed—for such a communication pattern. For a given direction of communication, we have as many communicators as there are communicating threads on the edge (a plane in 3D) since the operations of the threads are independent. The threads on a corner, however, use a single communicator for all directions since their operations for the different directions occur serially. The mapping of communicators to threads is not the same on each process. For example, thread 7 of the bottom-left process in Figure 4 must use a communicator for its north-south communication that is different from the communicator that thread 7 on the top-left rank uses for the same north-south direction. This difference in communicators prevent threads 1 and 7 on a process from using the same communicator and serializing their communication. In other words, given a map of communicators for the threads of a given process, the map for other processes can be derived by mirroring the map along the change in cartesian coordinates of the process. Listing 1 shows a 2D

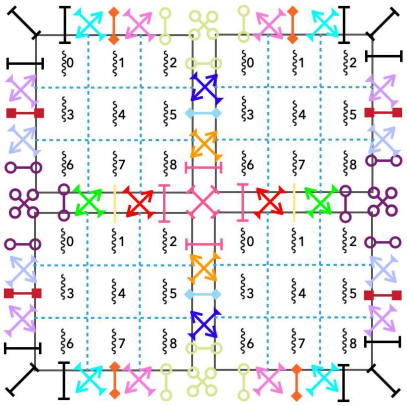

Fig. 4: Ideal communicator usage for a 2D 9-point stencil (stencils are the core kernels in hypre [54], Smilei [27], and Pencil [63]). Each box represents a process with 9 threads. Each thread has 1 patch. Each color-shape combination represents a communicator. Numbers represent thread IDs.

MPI+OpenMP 5-point stencil that exposes communication parallelism using communicators. For simplicity, it does not optimize communicator usage for corner threads. Lines 23–26 demonstrate the mirroring of communicator assignment to threads. This mirroring idea extends to the diagonal exchange in 9-point stencil as well: the user would need to extend lines 12–17 to create 4 additional sets of communicators—2 diagonals along the NS boundaries and 2 along the EW boundaries—each containing as many threads as there on the edge. Combining the mirroring strategy with the optimization to use a single communicator for corner threads will achieve the map illustrated in Figure 4 but at the cost of further complexity. The user would need to extend lines 23–24 to mirror the assignment of communicators to threads along both axes for all directions of exchange. Diagonal communicator sets require another dimension of mirroring that is exemplified by thread 1 on the bottom left process (see Figure 4) using the same communicator for its NE exchange as does thread 1 on the bottom right process for its NW exchange. Optimizing for reduced communicator usage is important for efficient use of network resources (see Lesson 3 below).

> *Lesson 2:* Using communicators to expose communication parallelism is not intuitive.

Continuing on the 2D 9-point stencil example, the intuitive approach to expose communication parallelism is to create as many communicators as there are threads and then use communicator i for thread i's send operations and communicator j for thread i's receive operations where j is the thread id of the remote thread that thread i is receiving from. This usage of communicators is correct, but it exposes *only half* of the available parallelism. The communication of adjacent threads on an edge occur in parallel but the operations of threads on opposite edges use the same communicator. For example, in Figure 4, thread 1's send operation uses communicator 1,

```
1  recv_from(proc_rank, tag, comm, *req, tid) :
2    if (need_mpi_op(tid)) :
3      MPI_Irecv(proc_rank, tag, comm, req)
4    else : /* use shared memory */
5  send_to(proc_rank, tag, comm, *req, tid) :
6    if (need_mpi_op(tid)) :
7      MPI_Isend(proc_rank, tag, comm, req)
8    else : /* use shared memory */
9  void main() :
10   /* px*py process grid with tx*ty local thread grid */
11   //  Create communicators to expose parallelism
12   for (i = 0; i < tx; i++) :
13     MPI_Comm_dup(COMM_WORLD, &ns_comm_a[i])
14     MPI_Comm_dup(COMM_WORLD, &ns_comm_b[i])
15   for (i = 0; i < ty; i++) :
16     MPI_Comm_dup(COMM_WORLD, &ew_comm_a[i])
17     MPI_Comm_dup(COMM_WORLD, &ew_comm_b[i])
18   /*Neighbor process ranks: n_rank,s_rank,e_rank,w_rank*/
19 #pragma omp parallel num_threads(N_THREADS)
20   {
21     /* Coords in local tx*ty thread grid: tid_x, tid_y */
22     // Choose the right communicator to use
23     n_comm = (ry%2) ? ns_comm_b[tid_x] : ns_comm_a[tid_x]
24     s_comm = (ry%2) ? ns_comm_a[tid_x] : ns_comm_b[tid_x]
25     e_comm = (rx%2) ? ew_comm_b[tid_y] : ew_comm_a[tid_y]
26     w_comm = (rx%2) ? ew_comm_a[tid_y] : ew_comm_b[tid_y]
27     for (iter = 0; iter < niters; iter++) :
28       recv_from(n_rank, tag_ns, n_comm, &reqs[0], tid)
29       /* recv_from [s_|e_|w_]rank with [s_|e_|w_]comm */
30       send_to(n_rank, tag_ns, n_comm, &reqs[4], tid)
31       /* send_to   [s_|e_|w_]rank with [s_|e_|w_]comm */
32       MPI_Waitall(8, reqs)
33       /* Compute after halo exchange */
34   }
```

Listing 1: 2D 5-point stencil using MPI communicators.

which thread 7 also uses for its receive operations.

> *Lesson 3:* Communicators have high network resource requirements to expose communication parallelism.

Even if the domain scientist achieves the ideal communicator usage, the number of communicators required to express communication parallelism is much higher than the minimum number of parallel channels required by the communication pattern itself. Such high resource requirements are concerning on today's many-core architectures. Consider the communication pattern of real-world stencil applications which is typically a 3D 27-point stencil (*e.g.*, hypre). Such applications decompose their domain into cubical patches. If $[x, y, z]$ represents the cubic arrangement of threads in an MPI process, the least number of communicators needed to express all of the available logical communication parallelism is $2xy + 2yz + 2xz + 8(xy + yz + xz - 1) + 4(xz + yz - z) + 4(xy + yz - y) + 4(xy + xz - x)$. The first three terms represent the directions perpendicular to the 6 faces, the fourth term represents the 8 corner diagonals, and the last three terms represent the edge diagonals. In terms of parallelism alone, however, the minimum number of parallel communication channels required is $xyz-(x-2)(y-2)(z-2)$ which is the number of threads communicating inter-node. If we consider a 64-core processor (*e.g.*, AMD EPYC Rome), the minimum number of communicators required to express communication parallelism is 808 (1 process per node with

[4, 4, 4] threads per process) which is over 14× higher than the minimum number of communication channels required by the 3D 27-pt stencil communication pattern. For this same pattern, other mechanisms use only as many resources as the communication pattern requires (see "Mechanism 3" below). The resource inefficiency of communicators can even hurt performance on some networks where the number of network hardware resources is limited (*e.g.*, 160 hardware contexts on Omni-Path [3]). Prior studies show hypre's communication time is over 2× higher with communicators than with other mechanisms on Omni-Path [68]. In such a scenario, the domain scientist is expressing all of the available communication parallelism and the application is using all of the network resources, but the observed performance benefit may not be as expected because of contention on the limited number of network resources which includes the software overheads of thread synchronization to access shared network queues [66].

> *Lesson 4:* Overloaded definitions of communicators can lead to mismatch in expected mapping to the underlying network parallelism.

A communicator has historically been viewed as a group of processes or as a means to isolate matching of messages. That it can double up as a means to express parallelism is a corollary of its definition. The multiple functions of a communicator can lead to a mismatch in expected mapping to the underlying network parallelism. For example, an application can initially create a set of communicators for grouping different processes and later use communicators to express parallelism. The MPI library underneath cannot differentiate between the two and could end up allocating a significant portion of the underlying network resources to the communicators used for grouping different sets of processes, leaving fewer network resources to map to for logical-parallelism-oriented communicators. MPI libraries can prevent this type of mismatch in expected mapping by introducing hints that allow an application to inform the library when it is creating communicators for the purposes of expressing logically parallel communication. But such hints would be implementation-specific.

> *Lesson 5:* The matching semantics of communicators limit communication parallelism for irregular and dynamic communication patterns.

As shown in Figure 4 and lines 23–26 in Listing 1, applications must ensure that the sending and receiving threads use the same communicator. This matching constraint is limiting for applications where the communication neighborhood of a thread changes over time, as it does in graph (*e.g.*, Vite [32]) and adaptive mesh refinement applications. This constraint also holds for applications running on modern task-based frameworks that exhibit irregular communication patterns. Figure 5 portrays this limitation for Legion [18] applications. Legion's event-based runtime maintains a receiving polling

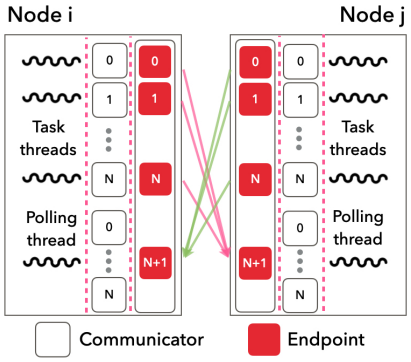

Fig. 5: Communicator vs. endpoints for Legion's runtime that implements an event-based system using multithreaded point-to-point communication.

thread per node to process incoming requests from the task threads on other nodes. The multiple task threads on a node can issue operations using distinct communicators, but the polling thread is forced to iterate over the communicators to process all incoming messages. Thus, on a single node, the polling thread conflicts with the communicators of the task threads. The polling thread relies on wildcards, and hence using partitioned operations for this communication pattern is not straightforward. With endpoints, on the other hand, the polling thread can use a distinct endpoint, use wildcards, and satisfy matching requirements. Prior evaluations show that Legion's polling thread processes events 1.63× slower with communicators than with endpoints [68].

Mechanism 2: Tags with hints. The new Info hints in MPI 4.0 allow domain scientists to use tags instead of communicators to express logical parallelism in applications that do not use certain MPI semantics (see Section II-A). Tags bypass some of the challenges with communicators, but introduce new ones. We describe the lessons learned from using tags below.

> *Lesson 6:* Using tags for communication parallelism is intuitive.

Most MPI+threads applications that use `MPI_THREAD_MULTIPLE` already encode thread IDs into the tags of their communication to differentiate operations that target different threads on the same target process (*e.g.*, hypre and Smilei), indicating that domain scientists intuitively think of tags as a means of expressing logical parallelism. Hence, the approach of using tags requires the least amount of changes to existing applications. These changes would only be in the form of creating a new communicator with Info hints that relax unneeded MPI semantics (see Listing 2).

> *Lesson 7:* Achieving optimal multithreaded communication performance with tags is tedious.

Even though tags and communicators have the same matching constraints, tags can provide more information. Consider the MPICH library that features multiple virtual commu-

```
/* Existing THREAD_MULTIPLE stencil apps */
#pragma omp parallel num_threads(N_THREADS)
{
  app_tag = src_tid << (NUM_TID_BITS + NUM_APP_BITS)
          | dst_tid <<  NUM_APP_BITS
          | app_tag;
  MPI_Send(dest_proc, tag, app_comm);
}
```

```
1 /* Leveraging parallelism info in tags of existing apps */
2 //(1) Relax unneeded MPI semantics with MPI 4.0 hints
3 MPI_Info_set(info,"mpi_assert_no_any_tag",   "true");
4 MPI_Info_set(info,"mpi_assert_no_any_source","true");
5 //(2) Achieve optimal mapping with MPI library hints
6 MPI_Info_set(info,"mpich_num_vcis",            N_THREADS);
7 MPI_Info_set(info,"mpich_num_tag_bits_vci", NUM_TID_BITS);
8 MPI_Info_set(info,"mpich_place_tag_bits_local_vci","MSB");
9 MPI_Info_set(info,"mpich_tag_vci_hash_type","one-to-one");
10 MPI_Comm_dup_with_info(app_comm, info, &tag_par_app_comm);
11 #pragma omp parallel num_threads(N_THREADS)
12 {
13   ... // (Tag encoding same as above)
14   MPI_Send(dest_proc, app_tag, tag_par_app_comm);
15 }
```

Listing 2: Exposing logically parallel communication through existing MPI tags in applications like Smilei and hypre.

nication interfaces (VCIs: network communication channels mapping to distinct network hardware contexts) [67], and the hypre library that encodes the IDs of the sending and receiving threads into the tag along with other application-related information. The thread IDs in the tag provide information about which local and remote VCI to use. But if MPICH does not know which bits of the tag encode communication parallelism information, then hypre is at the mercy of how MPICH hashes the tags into the multiple VCIs allocated to the communicator. Achieving the optimal mapping to VCIs requires hypre to inform MPICH which bits encode the sender's thread ID, which bits encode the receiver's thread ID, and how to map the bits to the underlying VCIs. For example, with a one-to-one mapping, MPICH can use the sender-thread-bits to map to a VCI on the host process and the receiver-thread-bits to decide which VCI to target on the remote process. Such intricate use of tags requires domain scientists to learn about the Info hints that are specific to an MPI library (see lines 6–9 in Listing 2).

> *Lesson 8:* Programs that use tags for communication parallelism may not be portable across MPI libraries.

As discussed above, achieving a high level of control and flexibility with tags requires a careful use of implementation-specific Info hints. It is likely that different MPI implementations will support tag-based and communicator-based mechanisms of exposing parallelism in different ways (VCIs are specific to MPICH; CRIs are specific to Open MPI [49]). Since HPC application developers are geared towards performance-oriented codes, they are compelled to adopt Info hints to optimally expose logically parallel communication to an MPI implementation. Hence, existing MPI mechanisms can result in reduced portability of codes which is highly undesirable.

> *Lesson 9:* Encoding communication parallelism in tags is limited by their existing use cases.

End users already use MPI tags for application-related information. Since the number of bits in a tag is limited, an application may not be able to encode further parallelism information into the tag. Encoding parallelism information with lesser-than-ideal number of bits is bound to hurt performance. Although we have not encountered first-hand an application that faces such a problem, others have reported running into tag-overflow issues on prominent applications (*e.g.*, SNAP [11], Smilei [10], and MITgcm [9]). Such reports indicate that applications already use a large portion of the tag space; encoding parallelism information into tags exacerbates the tag-overflow problem.

Mechanism 3: User-Visible Endpoints. Given their flexible interface (see Section II-B), endpoints combat the various concerns associated with communicators and tags to expose logically parallel point-to-point operations. Below, we delineate the lessons learned from mapping endpoints to different communication patterns.

> *Lesson 10:* Endpoints are intuitive to use.

Endpoints are an easier alternative to express communication parallelism even for patterns such as a 3D 27-point stencil since each local endpoint can flexibly address any other endpoint through a global endpoint rank. They are more intuitive to use than communicators because application developers are innately familiar with the semantics of traditional MPI ranks. Users express communication parallelism by communicating between endpoints as they do for MPI ranks in MPI everywhere programming. Lines 17–20 in Listing 3 shows this MPI-everywhere like addressing in a 2D MPI+OpenMP 5-pt stencil that exposes communication parallelism with endpoints. In fact, endpoints provide a level of flexibility beyond MPI everywhere: threads are not bound to an endpoint. In other words, endpoints do not enforce an association between threads and the data they work on; a thread is free to use any endpoint at any time. Thus, endpoints map well to tasking frameworks like OpenMP Tasks [25].

> *Lesson 11:* Endpoints distinguish between matching and parallelism information and thus apply seamlessly to all types of communication patterns.

Unlike existing MPI mechanisms, user-visible endpoints separate the task of expressing communication parallelism from the task of matching operations. Thus, using endpoints is straightforward even for irregular communication patterns, such as those of Legion applications (see Figure 5). Endpoints can flexibly adapt to dynamically changing communication neighborhoods because threads can address new remote endpoints while using the same local endpoint (lines 22–25

```
recv_from(ep_rank, tag, comm, *req, tid) :
  if (need_mpi_op(tid)) :
    MPI_Irecv(ep_rank, tag, comm, req)
  else : /* use shared memory */
send_to(ep_rank, tag, comm, *req, tid) :
  if (need_mpi_op(tid)) :
    MPI_Isend(ep_rank, tag, comm, req)
  else : /* use shared memory */
void main() :
  /* px*py process grid with tx*ty local thread grid */
  // Create as many endpoints as there are threads
  MPI_Comm_create_endpoints(COMM_WORLD, N_THREADS, info,
                            &comm_ep)
#pragma omp parallel num_threads(N_THREADS)
  {
    /* Coords in local tx*ty thread grid: tid_x, tid_y */
    n_ep = n_rank*N_THREADS + tx*(ty-1) + tid_x;
    s_ep = s_rank*N_THREADS + tid_x;
    e_ep = e_rank*N_THREADS + tid_y*tx  + tx-1;
    w_ep = w_rank*N_THREADS + tid_y*tx;
    for (iter = 0; iter < niters; iter++) :
      recv_from(n_ep, tag_ns, comm_ep[tid], &reqs[0], tid)
      /* recv_from [s_|e_|w_]ep with comm_ep[tid] */
      send_to  (n_ep, tag_ns, comm_ep[tid], &reqs[4], tid)
      /* send_to   [s_|e_|w_]ep with comm_ep[tid] */
      MPI_Waitall(8, reqs);
    /* Compute after halo exchange */
  }
```

Listing 3: 2D 5-point stencil using MPI endpoints.

in Listing 3). Additionally, endpoints do not suffer from overloaded definitions of existing MPI objects. Users do not need to compromise on the application's existing use of communicators or tags.

> *Lesson 12:* Endpoints lead to efficient resource usage and provide optimal mapping information without sacrificing portability.

By creating an endpoints communicator, users explicitly inform the MPI library that the new communicator is for the purposes of exposing communication parallelism. This information is baked into the API unlike the implementation-specific solution of using hints with communicators or tags. This is why endpoints do not suffer from high resource requirements either. In the prior example of executing hypre's 3D 27-point stencil on a 64-core processor, users need to create only as many endpoints as there are communicating threads, which is 56, 14.4× fewer than that required by communicators. Furthermore, the endpoints mechanism directly provides the MPI library with all the information needed to optimally map to the underlying network resources unlike the tag-based mechanism which requires the application to inform the MPI library about the specific tag bits that encode logical parallelism information. Since the optimal mapping information can be derived from a standardized interface, applications would be portable across MPI implementations.

Mechanism 4: Partitioned Communication. MPI 4.0 introduced the new semantics of partitioned operations for multithreaded point-to-point communication (see Section II-C). Like endpoints, partitioned operations promote portability, but their semantics can limit exposure of parallelism in irregular and dynamic communication patterns. We expand on our

```
test_recv_from(part_id, req, *rx_flag, tid) :
  if (need_mpi_op(tid)) :
    MPI_Parrived(part_id, req, rx_flag)
  else : /* use shared memory */
    *rx_flag = 1
send_to(part_id, req, tid) :
  if (need_mpi_op(tid)) :
    MPI_Pready(part_id, req)
  else : /* use shared memory */
void main() :
  /* px*py process grid with tx*ty local thread grid
     Each thread assigned to a tile_x*tile_y tile
     Neighbor process ranks: n_rank,s_rank,e_rank,w_rank*/
  // Create partitioned operations for parallel exchanges
  MPI_Precv_init(n_rx_buf, tx, tile_x, MPI_DOUBLE, n_rank,
                 tag_ns, COMM_WORLD, info, &reqs[0])
  /* MPI_Precv_init for s_rank with tx partitions
     MPI_Precv_init for [e_|w_]rank with ty partitions */
  MPI_Psend_init(n_tx_buf, tx, tile_x, MPI_DOUBLE, n_rank,
                 tag_ns, COMM_WORLD, info, &reqs[4])
  /* MPI_Psend_init for s_rank with tx partitions
     MPI_Psend_init for [e_|w_]rank with ty partitions */
  MPI_Startall(8, reqs)  // Activate all operations
#pragma omp parallel num_threads(N_THREADS)
  {
    /* Coords in local tx*ty thread grid: tid_x, tid_y */
    for (iter = 0; iter < niters; iter++) :
      send_to(tid_x, reqs[4], tid)
      /* send_to s_rank      with partition tid_x
         send_to [e_|w_]rank with partition tid_y */
    while (!n_rx_flag ||!s_rx_flag || !e_rx_flag
           || !w_rx_flag) :
       test_recv_from(tid_x, reqs[0], &n_rx_flag, tid)
       /* test_recv_from s_rank      with part tid_x
          test_recv_from [e_|w_]rank with part tid_y */
    /* Compute after halo exchange */
#pragma omp single
    { // Complete the requests before issuing next parts
      MPI_Waitall(8, reqs)
    } // required implicit thread barrier
  }
```

Listing 4: 2D 5-point stencil using MPI partitioned operations.

lessons learned with partitioned operations below.

> *Lesson 13:* Partitioned operations do not overload existing definitions, and they promote portability of codes.

Partitioned operations provide the same benefits as user-visible endpoints on two fronts: (a) they do not overload the definitions of existing MPI objects and hence minimize the mismatch in expected mapping to network parallelism; (b) they promote portability of applications across MPI implementations by their standardization. Unlike endpoints that build on top of existing MPI semantics, partitioned operations introduce new semantics. Given that research on partitioned operations is ongoing, the usability of partitions remains to be seen.

> *Lesson 14:* Partitioned semantics prevent threads from being completely independent.

The fundamental limitation of partitioned communication is that all threads (driving the multiple partitions in parallel) share the same MPI request. So, all threads would either contend on the MPI library's resources of the shared request or coordinate with each other to allow only a single thread to poll

for the completion of a partitioned operation. In either case, threads will incur contention or synchronization overheads before issuing their partition of the next message. Lines 37–40 in Listing 4 shows this synchronization requirement for a 2D MPI+OpenMP 5-pt stencil that exposes communication parallelism using partitioned operations. Application developers could use multiple partitioned operations (*e.g.*, double buffering) to dampen the overhead resulting from the semantic limitation, but they cannot eliminate them in a manner like the other two designs can. The implicit point of contention in the partitioned interface makes an application prone to the known high overheads of thread synchronization [15], [37], [43], [53]. It is not yet clear how the synchronization limitations of partitioned operations can be mitigated in modern applications where threads operate independently of each other [55], [60].

> *Lesson 15:* Persistence of partitioned operations prevent them from being used in dynamic and irregular communication patterns.

In dynamic communication where the destination of a message is not known apriori, using partitioned operations is a challenge since they are persistent by definition (lines 15–23 in Listing 4). Also, partitioned receive operations cannot use wildcards. Modern task-based runtimes (*e.g.*, Legion [60] and YGM [50]), however, have irregular communication patterns and rely on wildcards in their polling threads. Mapping partitions to the communication pattern in Figure 5 is challenging.

B. RMA communication

One-sided RMA have no matching semantics. Here, existing MPI mechanisms and user-visible endpoints are equally straightforward to use, but they each have unique concerns. The efficacy of partitioned operations for one-sided communication is yet to be studied.

> *Lesson 16:* Where the semantics of existing MPI mechanisms limit the exposure of logically parallel atomic operations, those of endpoints achieve optimal mapping of operations to the underlying network parallelism.

Using windows to expose communication parallelism constrains the parallelism information with MPI's atomicity semantics. This constraint limits the user from explicitly exposing logically parallel atomic operations within a single window even when the application does not need them to be ordered. Consider NWChem's get-compute-update pattern for its block-sparse matrix multiplication [57], [67] where a thread uses `MPI_Get` operations to retrieve the tiles it needs and, after the multiplication, uses an `MPI_Accumulate` operation to update the destination tile (see Figure 6). The `MPI_Accumulate` operations in a multithreaded process must use a single window for correct atomicity. Even though these parallel operations are independent, users have no way to explicitly expose this parallelism.

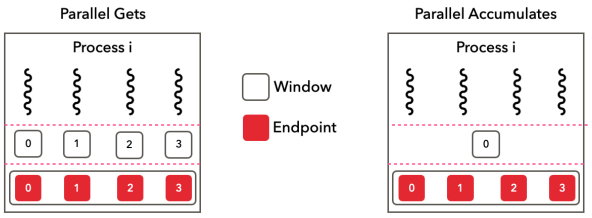

Fig. 6: MPI+threads communication parallelism in NWChem's block-sparse matrix multiplication.

The best they can do is relax MPI's ordering constraint (`accumulate_ordering=none`) and rely on the MPI library's hashing policies to map operations to parallel network channels. Any hashing policy, however, is prone to collisions and will prevent some operations from mapping to distinct network channels. With user-visible endpoints, on the other hand, users can use multiple endpoints within a single window to expose communication parallelism *and* maintain atomicity.

> *Lesson 17:* There exists preconceived notions in the MPI community about endpoints being direct handles to network resources.

A common misconception in the MPI community is to view endpoints as direct handles to network resources. This concern holds not just for RMA operations but also for point-to-point operations. As a result, the endpoints design is sometimes incorrectly regarded as a way for MPI libraries to dump the responsibility of managing network resources on the domain scientist which would in turn reduce the portability of applications. One explanation for this concern is the usage of the term "endpoints," which is typically associated with "network endpoints." The fact that user-visible endpoints were introduced for the purposes of utilizing network parallelism is likely to have exacerbated the misconception. User-visible endpoints are *not* handles to network resources, rather they are a means to flexibly express communication parallelism. Their usage is separate from the MPI library's task to map the exposed parallelism to the underlying parallel network resources. With the endpoints solutions, applications would create as many endpoints as there are streams of logically parallel communication. The MPI library would then funnel the streams of logically parallel communication on distinct network hardware contexts depending on their availability.

C. Collective communication

Research towards implementing collectives in a hybrid MPI+threads environment has primarily revolved around hierarchical algorithms where threads first implement the collective (*e.g.*, allreduce) amongst themselves and then one thread on each node participates in the internode collective [46], [69]. Recent studies using a fast MPI+threads library (Intel MPI 2019 [4]), however, demonstrate performance benefits when multiple threads drive a collective in parallel [20], [64]. Although existing MPI semantics require collectives

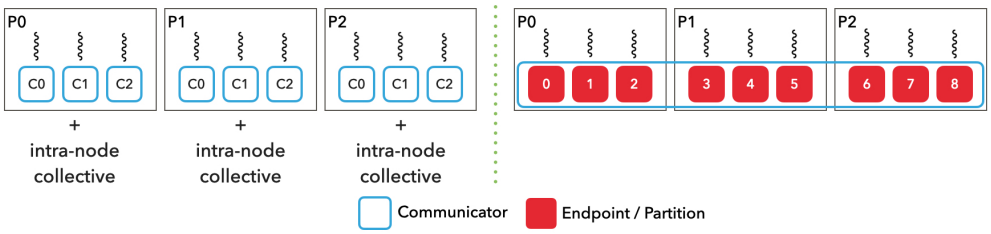

Fig. 7: Existing mechanisms vs. user-visible endpoints for collectives (*e.g.*, multithreaded Allreduce in VASP [64]).

on a communicator to be issued serially, applications may partition the collective-data of a process across threads and issue parallel collectives on the different data segments using a distinct communicator for each thread (*e.g.*, VASP collectives observe a speedup of over 2× with such an approach [64]).

> *Lesson 18:* Users need to perform the intranode portion of a collective with existing MPI mechanisms, but not with endpoints or partitioned operations.

With existing MPI mechanisms, users need to perform the intranode portion of the collective (see Figure 7). For example, in an allreduce collective, the user needs to perform a reduction step after all threads have completed the internode part of the allreduce. With user-visible endpoints or partitioned operations, on the other hand, the collective is only one step—all threads participate in a collective of the same communicator through different endpoints or partitions. The MPI library then conducts both the internode and intranode parts of the collective before returning from the operation. Although the performances of the two approaches are yet to be compared, we note that, from a design perspective, the endpoints and partitioned approaches are better because they do not force the user to manually handle the intranode portion of a collective. Arguably, shared-memory programming models feature direct support for collectives between threads; for example, OpenMP supports a reduction operation through compiler directives. But such support does not apply to all types of collective communication that MPI features (*e.g.*, `MPI_Alltoall`). A naïve implementation for such cases is likely to hurt performance for high thread counts (relevant on existing and upcoming many-core processors). Efficiently implementing a collective is not a trivial task; researchers have spent numerous efforts into optimizing collectives [22], [39]. Manually implementing tree-style and bucket algorithms is a tedious task that could instead be handled by the MPI library as is the case in the user-visible endpoints and partitioned operations designs.

> *Lesson 19:* Unlike existing MPI mechanisms and partitioned operations, user-visible endpoints lead to duplicated buffers on a node for some collectives.

The interface of user-visible endpoints results in duplication of data per node in cases where the result of the collective is the same across all ranks participating in the operation (*e.g.*, allreduce, broadcast, etc.). The destination buffers of the endpoints on a single process contain the same values (like MPI everywhere where each process features a copy of the resulting buffer) when only one such buffer is needed since all threads can directly read from the single buffer. With communicators, on the other hand, such duplication does not exist. For example, the user can perform an intra-node reduction into a single buffer that all threads can read from. The partitioned communication interface overcomes the duplication issue with endpoints. Each partition of a partitioned collective would be different sections of the input buffer. With threads driving distinct partitions, the MPI library would implement both the intranode and internode portion of a collective, and each process would host only one buffer that contains the result of the collective. We note, however, that we have yet to identify a case where an MPI everywhere application has run out of memory solely because of duplication of a collective's resulting data. Hence, the duplication of data in collectives with user-visible endpoints is not as concerning as the duplication of domain-level data that has caused applications to run out of memory with MPI everywhere. Furthermore, the duplication does not hold for collectives where the result of the collective is different for different ranks (*e.g.*, alltoall).

D. Heterogenous computing environments

The discussion in this paper so far has largely been in the context of (CPU-initiated) MPI+threads communication because the performance-oriented studies of the designs in the context of accelerators are yet to be conducted. Nevertheless, we briefly discuss how the different designs compare and apply to heterogeneous computing environments. Today's distributed applications that use accelerated computing have to largely rely on the control transferring back to the CPU from the accelerator (*e.g.*, GPU) before exchanging data with remote nodes. The system and runtime overheads (*e.g.*, GPU kernel launch latencies), however, limit the parallel efficiency of an accelerated application. One way to combat the scalability issue is to initiate communication from the accelerator. Technologies like NVSHMEM [42] and ROC_SHMEM [36] support GPU-initiated communication for the OpenSHMEM [23] programming model. GPU-initiated (point-to-point) MPI communication, however, remains an open problem. Executing MPI's matching engine on the GPU is known to be expensive [45].

> *Lesson 20:* Partitioned operations provide lightweight interfaces for device-initiated communication; the other two designs do not.

Out of the three designs, partitioned operations are best suited for high-speed device-initiated point-to-point operations. Through its non-critical-path `P[send|recv]_init` APIs, partitioned communication enables most of the serial overhead of setting up a low-level network message to be executed on a low-latency CPU core (before kernel launch) rather than a high-latency GPU compute unit. GPU thread blocks would then trigger or check for arrival of partitions with the lightweight `Pready` and `Parrived` operations [1]. Nonetheless, the limitations of partitioned operations described in Lessons 14 and 15 apply to heterogeneous computing scenarios as well—program control would need to return back to the CPU (*e.g.*, to execute an `MPI_Wait`) before the GPU can issue the next partitions of a message. Such repeated transfers of control will re-introduce device runtime overheads that device-initiated communication aims to address in the first place. Extensions that enable MPI operations to be enqueued into accelerator's work queues (similar to Nvidia's NCCL runtime [7]) may reduce such runtime synchronization overheads. These extensions, however, could apply to existing MPI objects and user-visible endpoints too.

Another way for applications to combat the device runtime overheads that hurt scalability is to use persistent GPU kernels that offload communication operations to the faster CPU cores through lightweight atomics or flags. How such an approach compares to device-initiated communication remains to be seen. Such application-level techniques are promising given the move towards system architectures with tightly integrated CPUs and GPUs where the latency to communicate between the two types of PUs will diminish [8], [61].

Furthermore, the benefits of device-initiated communication either compared to or in conjuction with techniques that leverage smart NICs or network hardware tag matching remains to be seen. All in all, the lessons from this paper remain relevant for heterogeneous computing environments moving forward.

IV. Meeting the Needs of Domain Scientists

"Rule of thumb for UX: More options, more problems."
— Scott Belsky

Table I shows a summary of the design choices to expose logically parallel communication for different types of MPI operations. With existing MPI mechanisms, users have to be aware of a multitude of options since each mechanism does not uniformly apply to all communication types and patterns. Domain scientists need to be aware about the mechanisms that become available when hints relax different semantics. Furthermore, each mechanism poses unique challenges: using communicators is hard because of MPI's matching semantics; the optimal use of tags is highly dependent on MPI implementation-specific hints; windows may not allow users to optimally expose the available communication independence.

The new partitioned communication interface also poses challenges to the user. The new semantics of partitions and the large expansion in MPI's API space indicate a multitude of options for the user to learn about and choose from. The interface is challenging to use for dynamic and irregular communication patterns especially those that use wildcards. More important, the semantic limitations of the interface prevent the user from achieving complete independence between threads.

One way to combat the limitations of the interfaces in MPI 4.0 is to design an abstraction on top of MPI that allows users to seamlessly expose communication independence in a user-friendly manner. The abstract layer would then use MPI 4.0 mechanisms underneath with MPI-implementation specific hints where needed (analogous to how different interconnects support the Open Fabrics Interfaces (OFI) [35] API through their own OFI providers). The challenge of such an abstraction is the design of an interface that applies to all communication patterns. The interface of user-visible endpoints is in fact an example of such a general abstraction; other forms remain to be investigated. But, more important, the implementation of any abstraction faces the semantic constraints of both existing MPI mechanisms (*e.g.*, matching semantics of communicators) and partitioned operations (*e.g.*, no wildcards).

In contrast, with user-visible endpoints, the interface that has not been standardized yet, users need to be aware of only one mechanism: *endpoints*, which applies uniformly to all types of MPI operations. Endpoints provide a flexible, straightforward interface for users to express logically parallel communication in a way that they are already familiar with (*i.e.*, using ranks). The concern with user-visible endpoints is a misimpression among domain experts about what endpoints represent. A reason for this is that the terminology in the MPI Endpoints proposal is oriented towards the community of MPI library developers. Since the ultimate goal of the proposal is to aid the domain scientist to express logically parallel communication, it is imperative that the proposal be user-facing.

To resonate with domain experts, we suggest rebranding[1] the proposal to *MPI Rankpoints* since it emphasizes that users can create multiple MPI ranks within a process. The goal is

[1]Rebranding techniques have proven to be successful with many technologies (*e.g.*, Android, Airbnb, etc.).

TABLE I: Summary of design choices to expose logically parallel communication (TBD: to be defined).

Operation	Existing MPI mechanisms	User-Visible Endpoints	Partitioned Communication
Point-to-point	Communicators or tags	Endpoints	Partitioned point-to-point APIs
RMA	Window(s)	Endpoints	Partitioned RMA APIs (TBD)
Collective	Communicators + user-driven intra-node collective	Endpoints	Partitioned collective APIs (TBD)

to educate and reinforce the understanding that *rankpoints* are not handles to network resources, rather they are a flexible, straightforward means of expressing parallelism that promotes portability of applications. While it requires one new API—`MPI_Comm_create_rankpoints`—it prevents the limitations of existing MPI mechanisms and partitioned operations. Rebranding is more than just a change in the name of the proposal. It requires a concerted effort by the MPI community in re-education through venues such as presentations and tutorials at flagship conferences and workshops.

V. Concluding Remarks

"People ignore design that ignores people."

— Frank Chimero

MPI+threads is a critical model to program the many-core processors of the current HPC clusters and the upcoming exascale systems. It is imperative that applications perform productively with it. The key to achieving high performance with MPI+threads requires effort from both MPI library developers and application developers. Recently, MPI libraries have made significant strides in this regard, and now the ball is in the court of domain scientists to expose communication parallelism to utilize the new fast MPI+threads libraries. Domain scientists, however, face many programming challenges (Lessons 1–5, 7–9, 14–16, 18) with the designs present in MPI 4.0 with respect to exposing the communication independence between threads. These solutions have their own merits (Lessons 6, 13, 20), but we show that they do not meet the needs of key communication patterns sufficiently. The MPI Rankpoints alternative, on the other hand, elegantly addresses the various limitations of the designs in MPI 4.0. MPI Rankpoints has its own challenges (Lessons 17 and 19), but its benefits (Lessons 10–12, 16, 18) prove to be a seamless option for domain scientists given that the design applies generally to all communication patterns. The lessons in this paper show that MPI Rankpoints, or a solution that is as flexible, warrants continued consideration as a viable solution for the MPI+threads programming model.

Acknowledgments

We thank the following individuals for their discussions and viewpoints on the different mechanisms of exposing logically parallel MPI communication: Damodar Sahasrabudhe from the University of Utah, Hengjie Wang from the University of California, Irvine, Julien Derouillat from Maison de la Simulation, Peter Mendygral from HPE Cray, Sayan Ghosh and Mahantesh Halappanavar from Pacific Northwest National Laboratory, Roger Pearce from Lawrence Livermore National Laboratory, and Hui Zhou and Pavan Balaji from Argonne National Laboratory. This work is supported by the National Science Foundation (NSF) under the award number 1750549.

References

[1] Accelerator Bindings for Partitioned Communication. https://github.com/mpiwg-hybrid/hybrid-issues/issues/4.

[2] Fujitsu Processor A64X. https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/.

[3] 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.

[4] Intel® MPI Multiple Endpoints Support. https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support.

[5] Latest draft of MPI Endpoints. https://github.com/mpi-forum/mpi-issues/issues/56.

[6] Legion's MPI backend. https://github.com/StanfordLegion/legion/tree/stable/runtime/realm/mpi.

[7] Nvidia Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[8] Nvidia Grace Hopper Superchip. https://www.nvidia.com/en-us/data-center/grace-cpu/.

[9] Tag overflow issue with MITgcm. http://mailman.mitgcm.org/pipermail/mitgcm-support/2014-October/009535.html.

[10] Tag overflow issue with Smilei. https://github.com/SmileiPIC/Smilei/issues/270.

[11] Tag overflow issue with SNAP. https://github.com/lanl/SNAP/issues/6.

[12] Tensorflow networking. https://github.com/tensorflow/networking/issues/17.

[13] A. Amer, C. Archer, M. Blocksome, C. Cao, M. Chuvelev, H. Fujita, M. Garzaran, Y. Guo, J. R. Hammond, et al. Software combining to mitigate multithreaded MPI contention. In *Proceedings of the ACM International Conference on Supercomputing*, pages 367–379. ACM, 2019.

[14] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+threads: Runtime contention and remedies. *ACM SIGPLAN Notices*, 50(8):239–248, 2015.

[15] A. H. Baker, R. D. Falgout, et al. Scaling hypre's multigrid solvers to 100,000 cores. In *High-Performance Scientific Computing*, pages 261–279. Springer, 2012.

[16] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *The International Journal of High Performance Computing Applications*, 24(1):49–57, 2010.

[17] P. Bangalore, A. Worley, D. Schafer, R. Grant, A. Skjellum, and S. Ghafoor. A Portable Implementation of Partitioned Point-to-Point Communication Primitives. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2020.

[18] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[19] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020.

[20] P. Boyle, M. Chuvelev, G. Cossu, C. Kelly, C. Lehner, and L. Meadows. Accelerating HPC codes on Intel (r) Omni-Path architecture networks: From particle physics to machine learning. *arXiv preprint arXiv:1711.04883*, 2017.

[21] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson. Distributed-memory breadth-first search on massive graphs. *arXiv preprint arXiv:1705.04590*, 2017.

[22] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[23] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3, 2010.

[24] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880*, 2018.

[25] B. R. de Supinski, T. R. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson. The ongoing evolution of openmp. *Proceedings of the IEEE*, 106(11):2004–2019, 2018.

[26] E. D. Demaine, I. Foster, C. Kesselman, and M. Snir. Generalized communicators in the message passing interface. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):610–616, 2001.

[27] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov, N. Aunai, et al. Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications*, 222:351–373, 2018.

[28] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 13–18. ACM, 2013.

[29] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible MPI endpoints. *The International Journal of HPC Applications*, 28(4):390–405, 2014.

[30] M. G. Dosanjh, A. Worley, D. Schafer, P. Soundararajan, S. Ghafoor, A. Skjellum, P. V. Bangalore, and R. E. Grant. Implementation and evaluation of MPI 4.0 partitioned communication libraries. *Parallel Computing*, 108:102827, 2021.

[31] I. Foster, C. Kesselman, and M. Snir. Generalized communicators in the message passing interface. In *Proceedings. Second MPI Developer's Conference*, pages 42–49. IEEE, 1996.

[32] N. Gawande, S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanaraman. Towards scaling community detection on distributed-memory heterogeneous systems. *Parallel Computing*, 111:102898, 2022.

[33] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, and A. H. Gebremedhin. Scalable Distributed Memory Community Detection Using Vite. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[34] R. E. Grant, M. G. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum. Finepoints: Partitioned multithreaded MPI communication. In *International Conference on High Performance Computing*, pages 330–350. Springer, 2019.

[35] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, et al. 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*, pages 34–39. IEEE, 2015.

[36] K. Hamidouche and M. LeBeane. GPU Initiated OpenSHMEM: Correct and Efficient Intra-Kernel Networking for dGPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 336–347, 2020.

[37] C. Hetland, G. Tziantzioulis, B. Suchy, M. Leonard, J. Han, J. Albers, N. Hardavellas, and P. Dinda. Paths to fast barrier synchronization on the node. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 109–120, 2019.

[38] E. Higgins, M. Probert, et al. Hybrid OpenMP and MPI within the CASTEP code. Technical report, ARCHER eCSE Technical Report, 2015.

[39] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations*, 1(2):58–75, 2014.

[40] D. Holmes. Introducing Endpoints into the EMPI4Re MPI library.

[41] D. J. Holmes, A. Skjellum, J. Jaeger, R. E. Grant, P. V. Bangalore, M. G. Dosanjh, A. Bienz, and D. Schafer. Partitioned collective communication. In *2021 Workshop on Exascale MPI (ExaMPI)*, pages 9–17. IEEE, 2021.

[42] C.-H. Hsu, N. Imam, A. Langer, S. Potluri, and C. J. Newburn. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–10. IEEE, 2020.

[43] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf. How many threads will be too many? On the scalability of OpenMP implementations. In *European Conference on Parallel Processing*, pages 451–463. Springer, 2015.

[44] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575, 2011.

[45] B. Klenk, H. Fröening, H. Eberle, and L. Dennison. Relaxations for high-performance message passing on massively parallel SIMT processors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 855–865. IEEE, 2017.

[46] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96, 2013.

[47] P. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. J. ONeill, C. Nolting, P. Edmon, J. M. Donnert, and T. W. Jones. WOMBAT: A scalable and high-performance astrophysical magnetohydrodynamics code. *The Astrophysical Journal Supplement Series*, 228(2):23, 2017.

[48] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl. Tackling exascale software challenges in molecular dynamics simulations with GROMACS. In *International conference on exascale applications and software*, pages 3–27. Springer, 2014.

[49] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm. Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019.

[50] B. Priest, T. Steil, G. Sanders, and R. Pearce. You've Got Mail (YGM): Building Missing Asynchronous Communication Primitives. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 221–230. IEEE, 2019.

[51] R. Rabenseifner, G. Hager, and G. Jost. 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*, pages 427–436. IEEE, 2009.

[52] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, et al. 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*, page 62. ACM, 2017.

[53] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on Intel Xeon Phi coprocessor. In *European Conference on Parallel Processing*, pages 588–600. Springer, 2015.

[54] D. Sahasrabudhe and M. Berzins. Improving Performance of the Hypre Iterative Solver for Uintah Combustion Codes on Manycore Architectures Using MPI Endpoints and Kernel Consolidation. In *International Conference on Computational Science*, pages 175–190. Springer, 2020.

[55] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. Berzins. Optimizing the hypre solver for manycore and GPU architectures. *Journal of Computational Science*, 49:101279, 2021.

[56] W. Schonbein, S. Levy, W. P. Marts, M. G. Dosanjh, and R. E. Grant. Low-cost MPI multithreaded message matching benchmarking. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 170–179. IEEE, 2020.

[57] M. Si, A. J. Pena, J. Hammond, P. Balaji, and Y. Ishikawa. Scaling NWChem with efficient and portable asynchronous communication in MPI RMA. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 811–816. IEEE, 2015.

[58] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multi-threaded MPI communication through a library-based implementation of MPI endpoints. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498. IEEE, 2014.

[59] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff. MPI at Exascale. *Proceedings of SciDAC*, 2:14–35, 2010.

[60] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel architectures and compilation*, pages 263–276, 2014.

[61] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, et al. Design and Analysis of an APU for Exascale Computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96. IEEE, 2017.

[62] H. Wang. *Algorithm Design for High-Performance CFD Solvers on Structured Grids*. PhD thesis, University of California, Irvine, 2021.

[63] H. Wang and A. Chandramolwishwaran. Pencil: A Pipelined Algorithm for Distributed Stencils. In *SC'20: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE/ACM, 2020.

[64] F. Wende, M. Marsman, J. Kim, F. Vasilev, Z. Zhao, and T. Steinke. OpenMP in VASP: Threading and SIMD. *International Journal of Quantum Chemistry*, 119(12):e25851, 2019.

[65] A. P. Worley. *A Portable Implementation of Partitioned Point-To-Point Communication Primitives*. PhD thesis, Tennessee Technological University, 2021.

[66] R. Zambre, A. Chandramowlishwaran, and P. Balaji. Scalable communication endpoints for MPI+Threads applications. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 803–812. IEEE, 2018.

[67] R. Zambre, A. Chandramowliswharan, and P. Balaji. How I learned to stop worrying about user-visible endpoints and love MPI. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–13, 2020.

[68] R. Zambre, D. Sahasrabudhe, H. Zhou, M. Berzins, A. Chandramowlishwaran, and P. Balaji. Logically Parallel Communication for Fast MPI+Threads Applications. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[69] H. Zhou, J. Gracia, N. Zhou, and R. Schneider. Collectives in hybrid MPI+ MPI code: Design, practice and performance. *Parallel Computing*, 99:102669, 2020.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Although this paper is not a performance-oriented study, it is motivated by performance studies that we had previously conducted (references 65, and 66 in the paper submission). We included results from our prior performance studies in a format relevant for this paper (for completeness) in Figure 1. These experiments in Figure 1 were all conducted on the Bebop cluster at Argonne National Laboratory on both of its bdw (Intel Broadwell processors) and knl (Intel Knights Landing processors) partitions. The servers in both partitions are connected with the Intel Omni-Path interconnect. Figures 1(a) (Artifact 1) and 1(c) (Artifacts 3 and 4) were run on Bebop's bdw partition, and Figure 1(b) (Artifact 6) was run on Bebop's knl partition.

The primary purpose of the attached artifacts is not the reproducibility of the performance studies published in our prior studies, but to back our claims of implementing the different mechanisms of exposing logically parallel communication in the various applications which is the main focus of our paper.

Our prior performance experiments were conducted during our design and development of a new MPI library (Artifacts 1 and 2) for MPI+threads, and during our performance evaluation of applications (Artifacts 3–8) using the new MPI library. The experiments related to MPI library designs were primarily conducted on the Skylake (Intel Skylake processors) and Gomez (Intel Haswell processors) clusters at the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. At the time of the performance experiments, the clusters featured different interconnects: Skylake servers were interconnected with Intel Omni-Path, and Gomez servers were interconnected with Mellanox InfiniBand (IB) EDR. NWChem's BSPMM (discussed in Figure 7 and Section III-B) was evaluated on JLSE's Gomez cluster. The hypre and Uintah applications (discussed in Section III-A) were evaluated on Bebop's knl partition (described above). The Legion-based Circuit application (discussed in Figure 5) was evaluated on Bebop's bdw partition (described above). The WOMBAT application was evaluated on the HPC3 cluster at the University of California, Irvine. The HPC3 cluster features Intel Skylake nodes that are interconnected using Mellanox IB EDR.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `https://doi.org/10.5281/zenodo.6941794`
Artifact name: Multi-VCI MPICH for fast MPI+threads (OFI netmod)

Artifact 2
Persistent ID: `https://doi.org/10.5281/zenodo.6941795`
Artifact name: Multi-VCI MPICH for fast MPI+threads (UCX netmod)

Artifact 3
Persistent ID: `https://doi.org/10.5281/zenodo.6941805`
Artifact name: Hypre linear solver with logically parallel communication

Artifact 4
Persistent ID: `https://doi.org/10.5281/zenodo.6941807`
Artifact name: Uintah using hypre with logically parallel communication.

Artifact 5
Persistent ID: `https://archive.softwareheritage.org/swh:1:dir:1303d1809238257a980ba5067a4f3e8f2cf40eb9`
Artifact name: WOMBAT with logically parallel communication

Artifact 6
Persistent ID: `https://doi.org/10.5281/zenodo.6941812`
Artifact name: Legion's MPI backend with logically parallel communication

Artifact 7
Persistent ID: `https://doi.org/10.5281/zenodo.6941820`
Artifact name: NWChem's BSPMM with logically parallel communication

Artifact 8
Persistent ID: `https://doi.org/10.5281/zenodo.6941816`
Artifact name: MPI+threads microbenchmarks with logically parallel communication

Reproduction of the artifact without container: The provided artifacts are MPI applications that use a modified MPI library. Hence, the primary requirements for building the artifacts are OFED libraries which involve kernel modules. Additionally, we no longer have access to the machines (Bebop and JLSE) on which the performance numbers for Figure 1 were collected.

More importantly, the primary contributions of our paper are not performance evaluations of applications, rather they are qualitative comparisons of the different mechanisms of exposing logically parallel communication in MPI+threads applications. The quantitative performance-oriented comparisons of the different mechanisms were published in our prior publications (references 65 and 66).

Regardless, here are the steps to reproduce the results in Figure 1(a).

1) Build Artifact 1 using '../configure –with-device=ch4:ofi:psm2 –with-libfabric=/path/to/your/libfabric/psm2-install –prefix=/path/to/your/install/ –enable-thread-cs=per-vci –enable-ch4-direct=netmod –with-ch4-max-vcis=X –enable-fast=O3 && make -j install'

2) Build pt2pt/multiple_isir_waitall_mbw_mr.c and pt2pt/single_isir_waitall_mbw_mr.c in Artifact 8 using 'make multiple_isir_waitall_mbw_mr' and 'make single_isir_waitall_mbw_mr' respectively.

3) To run the microbenchmark with 16 threads and logically parallel communication, use 'mpiexec -n 2 -ppn 1 -f ${HOSTFILE} -env MPIR_CVAR_CH4_OFI_MAX_VNIS 16 -env OMP_PLACES

cores -env OMP_PROC_BIND close -env HFI_NO_CPUAFFINITY 1 ./multiple_isir_waitall_mbw_mr -S 8 -T 16‘

4) To run the microbenchmark with 16 threads but without logically parallel communication, use ‘mpiexec -n 2 -ppn 1 -f ${HOSTFILE} -env MPIR_CVAR_CH4_OFI_MAX_VNIS 1 -env OMP_PLACES cores -env OMP_PROC_BIND close -env HFI_NO_CPUAFFINITY 1 ./multiple_isir_waitall_mbw_mr -S 8 -T 16‘

5) To run the microbenchmark with equivalent (16 cores) MPI everywhere parallelism, use ‘mpiexec -n 32 -ppn 16 -f ${HOSTFILE} -env MPIR_CVAR_CH4_OFI_MAX_VNIS 1 -bind-to core -env HFI_NO_CPUAFFINITY 1 ./single_isir_waitall_mbw_mr -S 8‘

The apps in both Figures 1(b) and 1(c) use the MPI library in Artifact 1 (see build instructions for it above). The build instructions for Uintah+hypre (Figure 1(b)) and Circuit Legion (Figure 1(c)) are in the READMEs of Artifacts 3, 4, and 6.

To run Uintah+hypre with the multi-VCI MPICH on 8 nodes with 1 process per node and 64 threads per process with logically parallel communication: ‘mpiexec -n 8 -ppn 1 -f ${HOSTFILE} -env HFI_NO_CPUAFFINITY 1 -env MPIR_CVAR_CH4_OFI_MAX_VNIS 64 -env OMP_NESTED true -env OMP_PROC_BIND "spread,spread" -env OMP_PLACES threads -env OMP_NUM_THREADS 64 -env HYPRE_TAG "6,12" ./${BINARY} -xthreads 4 -ythreads 4 -zthreads 4 -teamsize 1 8nodes_RMCRT_bm1_DO_solvertest.ups‘

To run Uintah+hypre with the multi-VCI MPICH on 8 nodes with 1 process per node and 64 threads per process without logically parallel communication: ‘mpiexec -n 8 -ppn 1 -f ${HOSTFILE} -env HFI_NO_CPUAFFINITY 1 -env MPIR_CVAR_CH4_OFI_MAX_VNIS 1 -env OMP_NESTED true -env OMP_PROC_BIND "spread,spread" -env OMP_PLACES threads -env OMP_NUM_THREADS 64 -env HYPRE_TAG "6,12" ./${BINARY} -xthreads 4 -ythreads 4 -zthreads 4 -teamsize 1 8nodes_RMCRT_bm1_DO_solvertest.ups‘

To run Uintah+hypre with MPI everywhere parallelism on 8 nodes with 64 processes per node: mpiexec -n 512 -ppn 64 -f ${HOSTFILE} -env HFI_NO_CPUAFFINITY 1 -env MPIR_CVAR_CH4_OFI_MAX_VNIS 1 -env OMP_NESTED true -env OMP_PROC_BIND "spread,spread" -env OMP_PLACES threads -env OMP_NUM_THREADS 1 -bind-to core ./1_mpi -npartitions 1 -nthreadsperpartition 1 8nodes_RMCRT_bm1_DO_solvertest.ups

To run Uintah+hypre with the multi-VCI MPICH on 8 nodes with 1 process per node and 64 threads per process: ‘mpiexec -n 8 -ppn 1 -f ${HOSTFILE} -env HFI_NO_CPUAFFINITY 1 -env MPIR_CVAR_CH4_OFI_MAX_VNIS 64 -env OMP_NESTED true -env OMP_PROC_BIND "spread,spread" -env OMP_PLACES threads -env OMP_NUM_THREADS 64 -env HYPRE_TAG "6,12" ./2_ep -xthreads 4 -ythreads 4 -zthreads 4 -teamsize 1 8nodes_RMCRT_bm1_DO_solvertest.ups‘

To run Circuit Legion on N nodes with logically parallel communication: mpiexec -n ${N} -ppn 1 -f ${HOSTFILE} -env MPIR_CVAR_CH4_OFI_MAX_VNIS 10 ./circuit -p ${N} To run Circuit Legion on 128 nodes without logically parallel communication: mpiexec -n ${N} -ppn 1 -f ${HOSTFILE} -env MPIR_CVAR_CH4_OFI_MAX_VNIS 1 ./circuit -p ${N}

Artifacts 3-7 contain READMEs that provide instructions on building the applications.