Logically Parallel Communication for Fast MPI+Threads Applications

Rohit Zambre, Damodar Sahasrabudhe, Hui Zhou, Martin Berzins, Aparna Chandramowlishwaran, and Pavan Balaji, *Senior Member, IEEE*

Abstract—Supercomputing applications are increasingly adopting the MPI+threads programming model over the traditional "MPI everywhere" approach to better handle the disproportionate increase in the number of cores compared with other on-node resources. In practice, however, most applications observe a slower performance with MPI+threads primarily because of poor communication performance. Recent research efforts on MPI libraries address this bottleneck by mapping logically parallel communication, that is, operations that are not subject to MPI's ordering constraints to the underlying network parallelism. Domain scientists, however, typically do not expose such communication independence information because the existing MPI-3.1 standard's semantics can be limiting. Researchers had initially proposed user-visible endpoints to combat this issue, but such a solution requires intrusive changes to the standard (new APIs). The upcoming MPI-4.0 standard, on the other hand, allows applications to relax unneeded semantics and provides them with many opportunities to express logical communication parallelism. In this paper, we show how MPI+threads applications can achieve high performance with logically parallel communication. Through application case studies, we compare the capabilities of the new MPI-4.0 standard with those of the existing one and user-visible endpoints (upper bound). Logical communication parallelism can boost the overall performance of an application by over 2×.

Index Terms—MPI+threads, MPI+OpenMP, exascale MPI, MPI_THREAD_MULTIPLE, MPI Endpoints, Uintah, hypre, Wombat, Legion

1 Introduction

Supercomputing applications are no longer able to run efficiently on modern distributed systems with the traditional MPI everywhere (typically one MPI process per core) programming model. The amount of memory per core has decreased over the past decade because of the disproportionate increase in the number of cores per processor compared with other on-node resources, such as memory, TLB space, and network registers [1], [2], [3]. MPI everywhere, on the other hand, has high memory requirements because the processes on a node host duplicated data [4]. The memory-hungry nature of MPI everywhere coupled with the decreased share of resources per core has resulted in applications running out of memory with large domains [5], [6]. To directly address this issue, the MPI Forum had introduced MPI Shared Memory [7] in MPI-3 (MPI+MPI), but such a solution is specific to sharing memory and is not a generic solution for sharing other on-node resources. Moreover, other aspects of MPI+MPI programming, such as on-node synchronization, can be more complex when compared with data parallel abstractions provided by alternative shared-memory programming models, such as OpenMP (e.g., dynamic load balancing with compiler directives) [8]. Hence, domain scientists have introduced a "new normal" in their programming model of choice: MPI+threads (typically an MPI process per node or NUMA domain, and a thread per core) [9]. MPI+threads (e.g., MPI+OpenMP) enables applications to efficiently share all of the limited resources of a node between cores while minimizing duplication of data on a node. As a result, many applications are now able to scale to large domain sizes on modern systems with MPI+threads [5], [10], [11].

MPI+threads is a critical model for programming modern processors in a scalable fashion; in practice, however, most applications tend to observe a slower performance compared with the application's MPI everywhere counterpart [5], [6], [12], [13]. MPI+threads programming raises many challenges over MPI everywhere. These include mitigating synchronization overheads of the shared-memory programming model [14], [15], [16], and preventing performance-degrading memory accesses (e.g., false sharing). But the most important challenge is the dismal communication performance of an MPI+threads application. The slow multithreaded MPI communication performance is a critical challenge to tackle since most scientific simulation campaigns run close to the strong-scaling limit where communication occupies a significant part of the application's runtime [17], [18], [19]. The MPI+threads version of the HYPRE solver [20], for instance, spends 2.81× more time in MPI than does its corresponding MPI everywhere version.

One of the reasons for poor multithreaded MPI communication (i.e., `MPI_THREAD_MULTIPLE`) performance is that state-of-the-art MPI libraries use conservative approaches, such as a global critical section, to maintain thread safety and MPI's ordering constraints. Recently, however, through the proactive use of the network parallelism available on modern interconnects, MPI libraries have made significant strides by demonstrating scaling `MPI_THREAD_MULTIPLE`

- *R. Zambre and A. Chandramowlishawaran are with the Department of Electrical Engineering and Computer Science, University of California, Irvine. E-mails: rzambre@uci.edu, amowli@uci.edu*
- *D. Sahasrabudhe and M. Berzins are with the Scientific Computing and Imaging Institute, University of Utah. E-mails: damodars@sci.utah.edu, mb@sci.utah.edu*
- *H. Zhou and P. Balaji are with the Mathematics and Computer Science Division, Argonne National Laboratory. Emails: zhouh@anl.gov , balaji@anl.gov*

communication performance that matches that of MPI everywhere [21], [22], [23]. If an MPI library is to funnel parallel MPI communication through distinct network channels, it needs to know which messages are unordered. Hence, a key requirement to reap the improved multithreaded MPI performance is the exposure of the independence between communication operations from parallel threads. The advanced MPI libraries are helpless if the application does not expose *logically parallel communication*—operations that are not subject to MPI's ordering constraints. Only the application can provide such information to the MPI library. State-of-the-art MPI+threads applications, however, typically do not express such logical communication parallelism.

A reason why domain scientists have not been exposing logical communication parallelism is that the semantics of the existing MPI-3.1 standard can either prevent the user from exposing all of the communication independence or make doing so cumbersome. With respect to combating these limitations, the MPI community holds two schools of thought. The earlier school of thought believes in providing domain scientists with user-visible endpoints to expose the logically parallel communication of an MPI+threads application [24], [25]. Endpoints are flexible interfaces and enable an application to express the maximum level of communication independence, but the challenge in introducing them is intrusive changes to the MPI standard in the form of new APIs [26], [27]. The more recent school of thought advocates a less-intrusive approach: MPI Info hints to relax the MPI semantics that are not needed by an application. Such relaxation allows domain scientists to use existing MPI mechanisms in a manner that matches the upper bound set by user-visible endpoints in terms of both performance and resource usage. In fact, the next iteration of the MPI standard, MPI-4.0, features new Info hints that provide domain scientists with new opportunities to express logical communication parallelism using existing MPI mechanisms.

In this paper, we showcase how domain scientists can eliminate the communication bottleneck in their MPI+threads applications using the different mechanisms of exposing logical communication parallelism. Using three application case studies, we compare the new opportunities of expressing logical communication parallelism in MPI-4.0 with the strengths and limitations of the existing MPI-3.1 standard and the upper bound set by user-visible endpoints. We study the Uintah computational framework, the WOMBAT astrophysics code, and the Legion programming system since they have been designed specifically to scale on upcoming exascale machines. We utilize and build on the fast MPI+threads library that we previously developed [21]. Unlike our previous work, we evaluate the impact of utilizing network parallelism on the end-to-end runtime of applications. Our evaluations show that the benefits of exposing logical parallelism go beyond the utilization of network parallelism. Removing the communication bottleneck enables applications to reap the maximum benefits of MPI+threads over MPI everywhere, significantly boosting performance (up to 2×) over MPI everywhere. Specifically, we make the following contributions.

(1) For the three MPI+threads application case studies, in Section 3, we present the bottlenecks in their design and performance and the challenges involved in addressing the bottlenecks.
(2) We provide a fast MPI+threads library (Section 5) that is capable of exploiting all the different mechanisms (described in Section 4) of expressing logically parallel communication. This is an extension of our previous work [21].
(3) In Section 6, we address the communication bottleneck in each case study and showcase how to unlock the full potential of the MPI+threads programming model.

2 Software and Testbeds

Our MPI implementation is based on the highly optimized CH4 [28] 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`). For most network operations, CH4 offloads functionalities, such as tag matching, to the low-level communication library, such as OpenFabrics Interfaces (OFI) [29] or Unified Communication X (UCX) [30]. When the network hardware cannot independently handle operations, CH4 falls back to using an active message implementation of the operation in its `ch4_core`.

We develop and evaluate our MPI implementation on the Intel Skylake and Intel Haswell testbeds at the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory (ANL). The Skylake nodes are interconnected with Intel Omni-Path (OPA) while the Haswell nodes are interconnected with Mellanox InfiniBand (IB) EDR. Since these testbeds comprise of a small number (< 10) of nodes, we use different, larger systems to evaluate the applications in this work. We evaluate the applications on the Bebop HPC cluster at ANL, and the HPC3 cluster at the University of California, Irvine. Bebop features a partition of Intel Knights Landing (KNL) nodes and a partition of Intel Broadwell nodes. Both partitions are interconnected by Intel OPA. The HPC3 cluster features Intel Skylake nodes that are interconnected using Mellanox IB EDR. For OPA, we use MPICH's OFI `netmod` in conjunction with PSM2 (OFI/OPA); for IB, we use the UCX `netmod` with Verbs (UCX/IB). For the same interconnect, the communication performance behaviors of the large-scale systems and the small-scale testbeds are similar.

We do not evaluate RMA operations on systems connected with Intel OPA since it emulates one-sided operations in software. This emulation hurts performance; our previous work analyzed the consequences of such emulation in detail [21]. Mellanox IB, on the other hand, provides direct hardware support for common RMA operations. Hence, we evaluate applications with RMA operations on systems using IB, and we use OPA-based systems only for applications with point-to-point operations.

3 Threads Hurting Applications

In this paper, we study applications and computational frameworks from three different domains: computational fluid dynamics (Uintah and HYPRE), astrophysics (WOMBAT), and data-centric programming systems (Legion). We discuss the distinct MPI+threads challenges associated with each case study.

3.1 Uintah and HYPRE

Since its inception at the University of Utah, the Uintah computational software framework has been used to solve a variety of fluid, solid, and fluid-structure interaction problems from diverse domains [31]. Its most notable application has been in the simulation of next-generation combustion problems to aid the design process of coal boilers. Boiler simulations enable engineers to build, test, and optimize designs that achieve a less-polluting coal burn. Uintah simulates the thermal radiation in such boilers using its ARCHES component, a three-dimensional large eddy simulation code that simulates heat, mass, and momentum transport in reacting flows using a low-Mach number approximation.

Two components play key roles in ARCHES' boiler simulations: Reverse Monte Carlo Ray Tracing (RMCRT) [32] that solves the radiation transport equation, and the HYPRE library [33] that solves a large system of linear equations projecting pressure at every timestep. RMCRT has been extensively developed to scale [34], [35], [36], and a key contributor to its scalability has been the adoption of the MPI+threads programming model. RMCRT is a memory-intensive algorithm since each process requires global domain information for the traversal of its share of rays through the entire domain. Given the data duplication per node with MPI everywhere, RMCRT runs out of memory for large domains. For example, Figure 1 shows that even with 1 patch per core, Uintah runs out of memory with MPI everywhere for a domain containing 16^3 patches (64 nodes) with 64^3 cells per patch. On the other hand, Uintah scales to much larger domain sizes with MPI+threads since MPI+threads dramatically reduces the copies of global data on a node [37]. But the performance of Uintah with MPI+threads is slower because the HYPRE library performs slower with MPI+threads than with MPI everywhere. Hence, although ARCHES needs MPI+threads to scale, it suffers from a loss in performance compared with MPI everywhere. In this work, we address the MPI+threads challenges in HYPRE so that ARCHES can achieve both high scalability and high productivity with MPI+threads.

The older version of HYPRE performed 3–8× worse than MPI everywhere, but recent efforts by Sahasrabudhe et al. [38] analyze the synchronization overheads of OpenMP in HYPRE and redesign the library so that multiple threads are able to call HYPRE in parallel with their respective patches. The threads asynchronously process their own patches inside HYPRE, similar to the way MPI processes process their patches in parallel in MPI everywhere. Unlike MPI everywhere, however, threads bypass MPI and directly use shared memory for intranode communication. Such restructuring eliminates all thread-synchronization overheads and all single-thread sections that previously existed in the HYPRE library. In this current version of HYPRE, each thread naturally conducts its own MPI communication in parallel with other threads (i.e., `MPI_THREAD_MULTIPLE`).

HYPRE's communication pattern is that of a 3D 27-point stencil. Although the communication of each thread is independent in this pattern, HYPRE still performs slower with MPI+threads than with MPI everywhere because of a higher MPI time (see Figure 2). HYPRE spends more time in MPI with MPI+threads because it does not expose

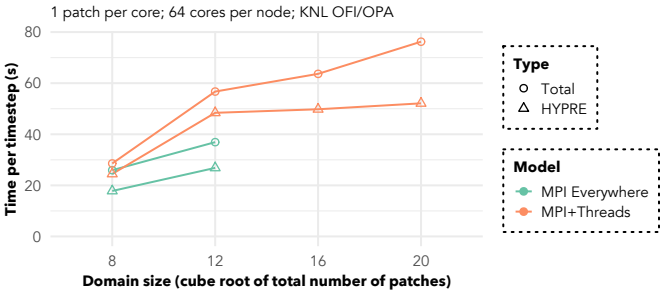


Fig. 1: MPI everywhere vs. MPI+threads in Uintah.

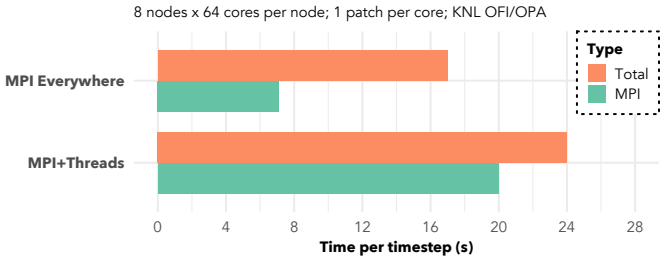


Fig. 2: Higher MPI time in MPI+threads HYPRE.

logical communication parallelism to the MPI library in a way that works for the existing MPI-3.1 standard. The MPI messages of all threads use the same MPI communicator, subjecting them to MPI's ordering constraints, such as the non-overtaking order.

HYPRE refrains from using multiple communicators because, at high thread counts, the number of communicators required for expressing all of the available independent communication can easily surpass the limited number of hardware contexts on the network (e.g., by over 5× with 64 threads on Intel OPA; see Section 6.1). Thus, even though HYPRE could expose communication parallelism with communicators, it would not practically achieve dedicated communication channels since the underlying network resources are limited.

HYPRE instead encodes the thread IDs of the sending and receiver threads into the MPI tag of the communication to differentiate operations targeting different threads on the destination process. The MPI library cannot exploit this encoded parallelism information in the tags because of the possibility of wildcards on receive operations. Even though HYPRE does not use any wildcards, the current MPI-3.1 standard does not provide any mechanisms for applications such as HYPRE to inform the MPI library that they do not use wildcards.

Since the MPI library does not observe any logically parallel communication in HYPRE, it funnels the communication from all threads through a single communication channel, serializing all of the otherwise independent communication operations. Even though the current version of HYPRE addresses many challenges of MPI+threads programming, the critical challenge of eliminating the communication bottleneck remains to be addressed. Successfully addressing this challenge by exposing logically parallel communication would unlock the true performance potential of MPI+threads for Uintah.

3.2 WOMBAT

Modern radio telescopes have provided us with new observations about the cosmological behaviors in the Universe. Given the complexity of the physical mechanisms involved in galactic interactions, interpreting these new observations is a challenging task. The CosmoPlasmas project, a partnership between academic institutions and HPE Cray, aims to better understand these new telescopic observations through numerical simulations that are designed to scale and run efficiently on modern high-performance computing systems [39]. The project participants have developed a new numerical framework, WOMBAT, that is geared to handle cosmological structure formation on the many-core architectures of exascale computing.

WOMBAT is a grid-based magnetohydrodynamic (MHD) code that simulates astrophysical phenomena to study the dynamics of highly conductive ionized astrophysical plasmas [40]. While numerous codes exist for astrophysical fluid simulations, they have not been adopted to run on the architectures of upcoming exascale machines. The primary guiding principle of WOMBAT, on the other hand, was to design an MHD code environment that scales well with high numbers of cores. It uses the MPI+threads programming model over MPI everywhere for many reasons: less expensive dynamic load balancing of computational tasks, a more symmetric workload between multithreaded processes, and more efficient use of the processor's shared resources. For example, Figure 3 shows that WOMBAT runs out of memory with MPI everywhere for large patch sizes but is able to run with MPI+threads because of the lesser amount of duplicated halo data.

WOMBAT's code structure features a single OpenMP parallel region as opposed to multiple OpenMP parallel-for loops. Each iteration of the simulation consists of different sections, and threads of a process collaboratively work on the computational and communication tasks of each section through dynamic load balancing. In terms of communication, WOMBAT uses MPI's RMA (`MPI_Put` and `MPI_Get`) operations to utilize the high-throughput, low-latency RDMA features of modern interconnects. In spite of these performance-oriented design choices, Figure 3 shows that WOMBAT achieves a lower scientific throughput with MPI+threads than with MPI everywhere, especially for smaller patch sizes. The primary reason for a slower MPI+threads performance is a higher boundary-data exchange time.

Threads in WOMBAT retrieve the boundary data for the patches of an MPI process in parallel (i.e., `MPI_THREAD_MULTIPLE`). Each thread first packs the boundary data of its local patch into a communication mailbox and signals readiness of this data to the corresponding MPI rank using an `MPI_Put` operation. A thread from the neighboring MPI rank then fetches the boundary data into its local mailbox using an `MPI_Get` operation. After unpacking the data into the patch's boundary zone, the thread signals the completion of its retrieval of the boundary data to the source MPI rank using another `MPI_Put` operation.

The communication of each thread in WOMBAT is independent, but WOMBAT does not explicitly express any of the independence in its multithreaded MPI communication—all operations use a single window. By default, not all operations on a window are unordered. Hence, due to the conservative approaches in existing MPI libraries, WOMBAT spends more time in MPI with MPI+threads than with MPI everywhere (see Figure 4). To eliminate the communication bottleneck in MPI+threads, WOMBAT needs to efficiently expose the logical parallelism in its multithreaded communication.

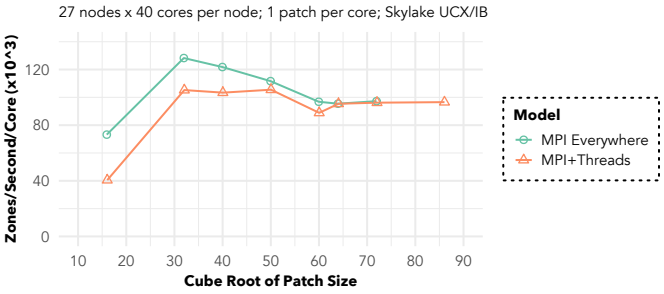


Fig. 3: MPI everywhere vs. MPI+threads WOMBAT.

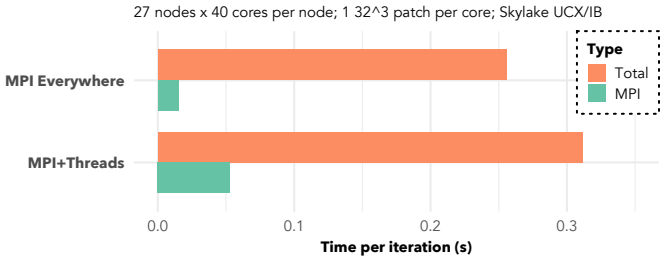


Fig. 4: Higher MPI time in MPI+threads WOMBAT.

3.3 Legion

In modern heterogeneous computing environments, the cost of data movement dictates the overall computational efficiency of an application. Hence, to achieve high performance, application developers need to allocate and move data efficiently, in addition to expressing parallelism. More important, they need to do so for every new architecture the application needs to run on. The penalty of not doing so is very poor performance. Legion, a data-centric parallel programming system, reduces this programming burden on domain scientists [41]. It is targeted for writing portable high-performance applications to run on distributed heterogeneous architectures. Given its advantages of automated data movement mechanisms and user-controlled mapping to architectures, Legion has become a prominent programming model for HPC applications.

The low-level runtime providing portability to the Legion system is Realm [42]. The Realm interface provides primitives that can be implemented on a wide range of technologies including a variety of high-performance interconnects and GPUs. Since Legion applications describe dependencies between operations, Realm is able to exploit opportunities to overlap operations in scenarios that are too difficult for the domain scientists to express by themselves and thus obtain higher performance. Realm uses an event-

based system to dynamically represent the control graph of a parallel program described by the Legion runtime. Such an event-based system allows Realm to flexibly execute strategic mappers of different architectures as well.

The Realm runtime is multithreaded and is implemented by using Pthreads. The internode communication of this multithreaded runtime has been designed around GASNet's active messages [43]. Active messages from the multiple task threads of a node contain a command, a payload, or both (for small payloads) for the target node, which, upon receiving the message, processes the payload through a corresponding handler. Recent efforts from Argonne National Laboratory implement an MPI module for Realm's internode communication. These efforts are in line with Legion's mission of portability and high performance since MPI, compared with GASNet, is more widely supported and adopted to obtain high performance on HPC systems. The MPI backend implements Realm's active-message-style communication using MPI's point-to-point operations. Like the GASNet backend, it maintains a single polling thread on each node to process the incoming messages from other nodes. While there can be multiple task threads initiating active messages (i.e., `MPI_THREAD_MULTIPLE`), the theoretical bottleneck in communication performance is the single receiving polling thread. More important, communication is a major constituent of Legion applications especially at the limits of strong scaling [41], [44].

Since the communication of each parallel thread is independent, in theory each thread's operations could funnel through a distinct network context. However, the current MPI backend does not expose the available parallelism. All threads issue their operations on `MPI_COMM_WORLD`. Fully exposing the logically parallel communication, however, is not possible with the existing MPI semantics. Although the task threads can send messages through distinct communicators, the receiving polling thread cannot use its own communicator. The matching semantics of MPI force the polling thread to iterate over the communicators to check for all incoming messages. Thus, on a single node, the polling thread uses the same communicators as those of the task threads. Such usage of communicators does not express all of the available logical parallelism in Legion's internode communication. In order to unlock the performance potential of Legion applications, it is important that the polling thread does not contend with other threads and that it gets its own dedicated communication channel to the network.

4 The Two Schools of Thought

Exposing logically parallel communication is a critical challenge in all three application case studies of this work. How can MPI+threads applications expose communication independence so as to map to the parallel network channels? In this regard, two schools of thought exist in the MPI community. Both agree on a critical precursor: MPI libraries must implement parallel communication channels, each mapping to a distinct network context. The two ideologies differ, however, in the mechanisms of expressing logical parallelism. We describe the two schools of thought below.

4.1 User-Visible Endpoints

To help with the expression of logical parallelism, researchers initially proposed to extend the MPI standard to introduce user-visible MPI Endpoints [27]. New APIs would allow the user to create communicators and windows with multiple endpoints [26]. Each endpoint would take on the semantics of an MPI rank and be directly addressable, giving users explicit control over the communication between endpoints. If the user mapped each thread to a distinct endpoint, then all threads could have dedicated communication paths to the network. We note, however, that endpoints are not handles to network resources; rather, they are a means to express logically parallel communication. The MPI library would map the communication from different endpoints to its internal communication channels, which could be fewer than the number of endpoints depending on the availability of network resources.

Given their flexible interface (ability to specify both the local endpoint to issue an operation from and the remote endpoint to target), user-visible endpoints represent the upper bound in expressing the communication parallelism available in an application. Several efforts indeed show scaling communication throughput with user-visible endpoints through multiple communication channels inside the MPI library [24], [25], [45], but the very nature of flexible communication control requires intrusive extensions to the standard in order to introduce the new concept of endpoints to MPI users.

4.2 Existing MPI Mechanisms

More recent efforts advocate using existing MPI mechanisms such as communicators, tags, and windows to express logically parallel communication [21], [22]. This ideology stems from the fact that the existing MPI standard already allows users to express parallelism in their communication. An efficient MPI library could then map the user-expressed parallelism to its internal communication channels. We describe the mechanisms to express parallelism in the existing MPI-3.1 standard and the upcoming MPI-4.0 standard.

4.2.1 MPI-3.1

The existing MPI standard allows logically parallel communication for both point-to-point and RMA operations [21].

Communicators. Point-to-point operations in distinct communicators can never match with each other and hence are fully independent. Thus, users can use distinct communicators to express the communication independence between point-to-point operations from parallel threads.

Windows. All types of RMA operations do not maintain any ordering of operations across windows. Although nonatomic operations are unordered regardless of whether or not they are on different windows, users should 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 constraints and explicitly expose parallelism for any type of RMA operation, users have the option of mapping operations from distinct threads though different windows.

4.2.2 MPI-4.0 draft

Some of the existing semantics of MPI-3.1 prevent applications from using certain mechanisms to expose logically parallel communication. For example, MPI-3.1 allows point-to-point parallelism to be expressed only through communicators because of the possibility of wildcards (e.g., `MPI_ANY_TAG`) within a communicator. Such semantics can be limiting to the application especially when the application does not rely on the non-overtaking order or wildcard features of MPI. To overcome such limitations in the next iteration of the standard, MPI-4.0, the MPI Forum has voted in new MPI Info hints that will allow an application to relax the MPI semantics it does not need. Not only does such relaxation of semantics yield new opportunities for performance optimizations, but it also opens up new opportunities for the application to express logically parallel communication.

Tags with hints. Relevant Info hints from the draft MPI-4.0 standard include `mpi_assert_no_any_source`, `mpi_assert_no_any_tag`, and `mpi_assert_allow_overtaking` [46]. The first two, when set, inform the MPI implementation that the application will not use wildcards on the given communicator. The third informs the MPI library that the operations do not need to be matched in the order that they were posted (relaxing the non-overtaking order). These hints mean that users can express logically parallel communication within a communicator by using a distinct tag for each thread. While the matching semantics of MPI still apply, a tag-based parallelism approach is more tractable than a communicator-based one since existing MPI+threads applications using `MPI_THREAD_MULTIPLE` tend to already encode communication parallelism information in their MPI tags [11], [38].

5 A Fast MPI+Threads Library

Both schools of thought regarding the expression of logical communication parallelism rely on the fundamental idea that the MPI library must maintain parallel communication channels, each mapping to a distinct network context. In this section, we summarize our previous work that establishes fast parallel communication channels inside the MPI library using virtual communication interfaces (VCIs). In addition, we evaluate and discuss the multi-VCI infrastructure in the context of the draft MPI-4.0 standard. The multi-VCI MPICH library designed and used in this work is open source for both the OFI [47] and UCX [48] netmods.

5.1 A Virtual Communication Interface

A VCI is an abstract representation of a communication channel. Each VCI maps to a distinct network context and contains its own independent set of communication resources: a transmit queue and a receive queue to issue operations and a completion queue to poll for progress of issued operations. These resources maintain a FIFO order of the MPI operations that map to it. The physical realization of a VCI depends on the `netmod` and the underlying interconnect. A VCI in the OFI `netmod` is an OFI endpoint bound to an OFI completion queue. For Intel OPA, the OFI endpoint maps to a hardware context on the Intel HFI network adapter [49]. 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 [50].

5.2 Fast Parallel Communication Channels

With a pool of VCIs, we establish parallel communication channels in the MPI library. Since threads need to be able to access the VCIs in parallel, fine-grained critical sections are an important precursor to extract the benefits of VCIs. Balaji et al. [51], [52] and Amer et al. [53] have designed fine-grained critical sections for an MPI library by splitting the responsibility of the global lock such that each class of objects is protected by its own lock. We extend their designs such that the resources of each VCI are protected by a lock belonging to the VCI. In this way threads that map to different VCIs can access the VCIs without contention.

Simply employing multiple VCIs in an MPI library with fine-grained critical sections, however, yields no performance benefit (see Figure 5). Together they perform similarly to the original version of the MPI library that employs a global critical section and uses a single VCI. To enable a fast MPI+threads library, we need to restructure the internals of the MPI library to provide contention-free paths to threads mapped to different VCIs. We identify three sets of optimizations that eliminate bottlenecks present in the architectures of MPI libraries.

Per-VCI progress. The first optimization deals with the progress of multiple VCIs during operations such as `MPI_Waitall`. The naïve approach would be to blindly poll for progress on all VCIs (global progress) during a progress function, but such an approach hurts performance drastically especially when multiple threads progress operations in parallel. What we need is per-VCI progress: progress functions that poll for progress only on the VCI associated with the operation. There are, however, cases of correct MPI programs (examples in [21]) where pure per-VCI progress leads to a deadlock because the program relies on shared progress between threads—thread A needs to progress the operations of thread B that were issued on VCI B. To accommodate for such communication patterns, the MPI library must execute global progress to ensure correctness of the implementation. We therefore adopt a hybrid per-VCI progress model where we first poll only the VCI of the operation, and if progress has not been made after a certain number of attempts, we switch to one round of global progress. Figure 6 shows that 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. MPI libraries typically maintain a global memory pool for requests. Thus, even when operations from multiple threads map to different VCIs, they contend on the lock of the global request pool 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 employ as the main optimization a per-VCI request cache, which is a cache of requests

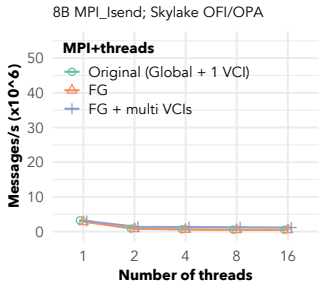


Fig. 5: Multiple VCIs.

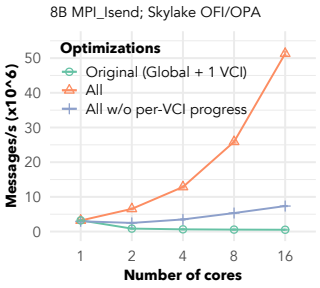


Fig. 6: Progress opts.

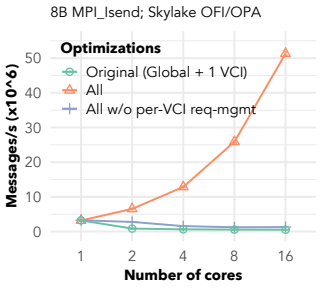


Fig. 7: Request opts.

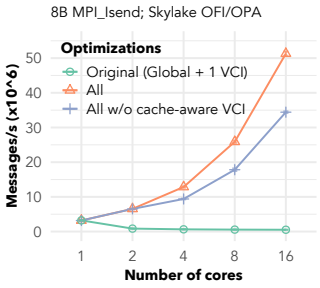


Fig. 8: Cache-aware VCI.

from the global pool. Access to each per-VCI request cache is protected by the lock of the VCI. Thus, in the common case, threads mapped to different VCIs do not contend on the lock of global request pool; instead, they acquire and release requests to the VCI's cache of requests. Figure 7 shows the benefits of the per-VCI request management. Without it, throughput is $39.98\times$ lower (All w/o per-VCI req-mgmt) compared with all optimizations.

Per-VCI cache-line awareness. As is the case with any multithreaded code, the multi-VCI infrastructure is prone to effects of false sharing. Threads mapped to consecutive VCIs could witness false sharing between the locks of the VCIs; hence, we use compiler attributes to cache-align each VCI. Figure 8 shows that without a cache-aware VCI, the message rate is $1.49\times$ lower (All w/o cache-aware VCI).

Summary. Fine-grained critical sections, multiple VCIs and all the above optimizations together enable a fast MPI+threads library. Without any of these features, multithreaded communication performance is dismal.

5.3 Mapping Logical Parallelism to Network Parallelism

As we saw in Section 4, users can express logically parallel communication either through user-visible endpoints or through existing mechanisms. In the endpoints mechanism, we map the communication from the distinct endpoints to distinct VCIs. This mapping gives users full control over communication between endpoints. Since users use existing MPI objects such as communicators and tags for other purposes apart from expressing communication parallelism, the possibility of a mismatch in expected mapping to VCIs is more likely with existing MPI mechanisms than with user-visible endpoints [21].

To allow users to better control the mapping of their logically parallel communication to VCIs, we introduce a new set of MPI Info hints complementing those introduced in the draft MPI-4.0 standard. The new set of hints does not influence any MPI semantic; rather, it is specific to our MPI implementation.* The hints allow domain scientists to relay application-specific communication parallelism information to the MPI library to achieve the best mapping of the logically parallel communication to the underlying network parallelism. By default, if the user provides no MPI Info hints, all communicators and windows map to the same single VCI. By using hints, however, applications can request a new VCI or multiple new VCIs for a communicator or window.

In point-to-point communication, if the user requests a single VCI for a communicator, all operations using that communicator will funnel through the VCI mapped to that communicator. This allows users to expose logically parallel communication using communicators. If communicators are insufficient to expose parallelism in an application's communication pattern, domain scientists may request multiple VCIs for a communicator and supply the appropriate MPI-4.0 hints (e.g., `mpi_assert_no_any_tag`) to expose the communication parallelism within a communicator using, for example, tags. We provide Info hints for users to inform the MPI library which bits of the tag they will use to express logical parallelism. In this way the MPI library can optimize the mapping of tags to VCIs by using only the reserved tag bits hinted by the user. Note that these hints do not break MPI's matching requirements. In any case, the user must ensure that the ⟨comm, rank, tag⟩ envelopes of operations match for successful communication to occur.

To express RMA communication parallelism with existing MPI mechanisms, users have the option either to let the MPI library automatically exploit their multithreaded RMA operations (non-atomic only) or to explicitly express the communication parallelism using separate windows (wherever the MPI semantics allow them to do so). The nature of automatic mapping, however, suffers from fundamental performance and semantic limitations. For example, an automatic mapping that hashes the thread ID to the limited network resources suffers from hash collisions. More important, as we described in Section 4.2.1, mixing synchronization and initiation operations on the same window is tricky. Hence, we focus our analysis on the explicit option of expressing parallelism with windows. As is the case with communicators, users can request a new VCI for a window through a hint.

Fallback mechanisms. In either mechanism of expressing communication parallelism, the VCI pool may be empty during the creation of communicators, windows, or endpoints since the number of contexts on the network hardware is limited (e.g., Intel OPA features only 160 hardware contexts on the HFI adapter [49]). For such cases, the MPI library must maintain fallback mechanisms. For example, once all VCIs have been allocated, the library could use a round-robin approach over the already allocated VCIs for any new communicators or windows requesting new VCIs. When the user frees a window or communicator, its associated VCIs are returned to the pool.

*. Introducing new implementation-specific Info hints does not violate the MPI-3.1 standard.

5.4 Microbechmark Evaluation

In this section, we demonstrate the performance of the fast MPI+threads library on communication-intensive microbenchmarks using the different mechanisms of expressing parallelism. The benchmarks demonstrate 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.

For MPI+threads, we compare the different mechanisms of expressing communication parallelism against the state of the art where either the user does not express logical parallelism or the MPI library conservatively maintains MPI's ordering constraints using a global critical section and a single communication channel. Our previous study shows that in either case of the state of the art in MPI+threads, communication throughput does not scale [21]. We also compare against the transparent utilization of network parallelism in MPI everywhere. In MPI+threads, we spawn one rank per node with an OpenMP thread per core; MPI everywhere uses one rank per core.

5.4.1 Point-to-Point

With the existing MPI-3.1 standard, users can express logically parallel communication using a distinct communicator for each host-target thread pair. With MPI-4.0, users may also use communicators, but they have the additional option of relaxing the unneeded semantics of MPI and using tags to express communication parallelism within a single communicator. They can do so by using a distinct tag for each communicating thread-pair. In the microbenchmark, for example, encoding the thread IDs of the sending and receiving threads into the tag of the MPI message exposes the logically parallel communication to the MPI library. With user-visible endpoints, each thread uses its own endpoint and directs its communication to the target remote endpoint.

For the different mechanisms of expressing parallelism, Figure 9 shows the message-rate scalability of a small-message `MPI_Isend`, and Figure 10 shows the message rate of `MPI_Isend` with 16 cores across varying message sizes on OFI/OPA. The communication throughput scales equally well for all different mechanisms of expressing parallelism. Even with exposed communication parallelism, however, MPI+threads performs slower than the corresponding MPI everywhere version. The level of network utilization is the same in both cases, but MPI+threads incurs overheads of thread safety over MPI everywhere even in the uncontended case. These overheads include lock acquisitions, atomics for completion, and reference counting [21].

5.4.2 RMA

We express communication parallelism in the microbenchmark with existing MPI mechanisms using a distinct window per thread. With user-visible endpoints, all threads issue operations on the same window but each uses a distinct endpoint.

Similar to Figures 9 and 10, Figures 11 and 12 demonstrate, for the different mechanisms of expressing RMA communication parallelism, the throughput scalability of a small-message `MPI_Put` and the 16-core message rate of `MPI_Put` across varying message sizes, respectively, on UCX/IB. Mellanox InfiniBand implements most contiguous RMA operations completely in hardware; hence, the communication throughput scales equally well with both windows and endpoints.

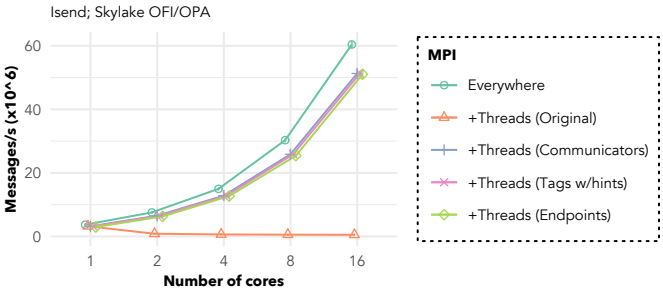


Fig. 9: Message-rate scalability of 8-byte `MPI_Isend`.

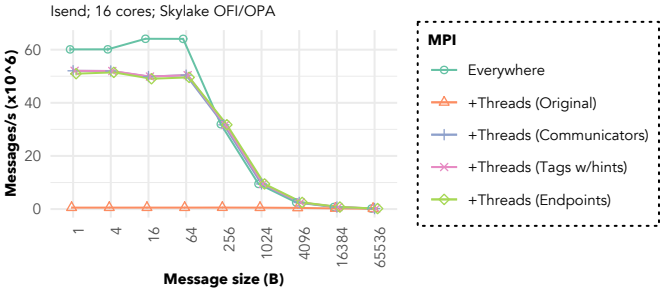


Fig. 10: `MPI_Isend` throughput with varying message sizes.

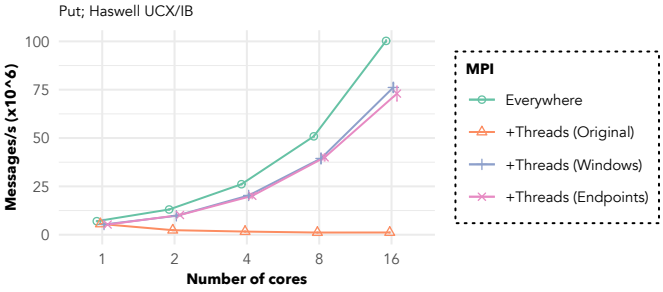


Fig. 11: Message-rate scalability of 8-byte `MPI_Put`.

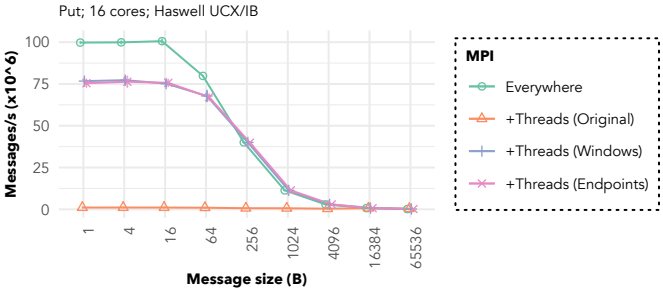


Fig. 12: `MPI_Put` throughput with varying message sizes.

6 Logical Communication Parallelism for MPI+Threads Applications

In this section, we address the challenges of the three applications (described in Section 3) by exposing logical communication parallelism. For each application, wherever possible, we compare the performances of MPI everywhere (a rank per core) and MPI+threads (a rank per node/socket; a thread per core) parallelism. For MPI+threads, we discuss the ways in which we express parallelism through existing MPI mechanisms and compare their performances with those of the original version of the MPI library and the upper-bound set by user-visible endpoints.

6.1 Uintah and HYPRE

In Section 3.1, we learned that the MPI+threads version of Uintah's ARCHES component suffers from a loss in performance because the HYPRE library does not expose logical communication parallelism to the MPI library. So, we first evaluate the different mechanisms of exposing logical parallelism in HYPRE on Bebop's KNL OFI/OPA cluster. HYPRE decomposes its domain in a cubical fashion and distributes its cubical patches (each patch in our evaluation contains 64^3 cells) between cores statically for both MPI+threads and MPI everywhere parallelism.

User-visible endpoints. We can express maximal communication independence with user-visible endpoints by creating as many endpoints as there are threads. Each thread uses its local endpoint to issue operations, and it communicates with the remote thread using the rank of the target endpoint. On a KNL node with 1 MPI process and 64 threads, 64 endpoints per node exist. Since the underlying OPA network features 160 hardware contexts, user-visible endpoints do not exhaust the network resources.

Communicators. Expressing the communication parallelism within the confines of the existing MPI-3.1 standard is theoretically possible using communicators. However, the communicator-based approach is complex for a 27-point 3D stencil because of MPI's matching constraints—both the sender and receiver thread must use the same communicator. To express parallelism with this constraint, we need as many communicators as there are threads simultaneously participating in MPI communication for each direction. If we consider $[x, y, z]$ to be a vector representing the cubic arrangement of threads in an MPI process, the least number of communicators we need 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. For a KNL node with $[4, 4, 4]$ threads, we need at least 808 communicators each with a distinct VCI for contention-free communication. Such use of network resources is highly inefficient since the number of VCIs required is over $12\times$ higher than that required by user-visible endpoints. Additionally, since the number of network hardware contexts on OPA is limited to 160, the MPI library is forced to funnel the communication from independent communicators through the same network channels. For our evaluation, we use a round-robin policy to assign VCIs to communicators. Arguably, if the limit of underlying network resources was higher (e.g., 8k on Mellanox IB), a round-robin policy would not come into effect, but the problem of inefficient resource usage would still remain.

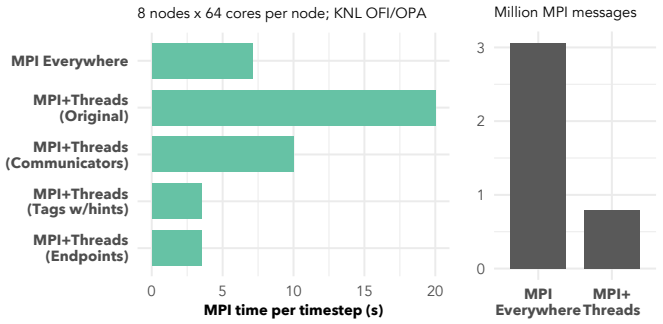


Fig. 13: Different mechanisms of exposing logical communication parallelism for HYPRE (1 patch per core).

Fig. 14: MPI communication volume per node.

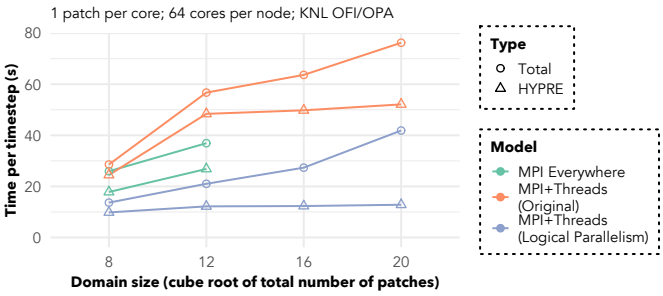


Fig. 15: Uintah with logically parallel communication.

Tags with hints. A less complex way to expose logical parallelism using existing MPI mechanisms is to leverage the parallelism information that HYPRE already encodes in its MPI tags to differentiate matching information for messages targeting different threads on the same target process (see Section 3.1). What prevents MPI libraries from mapping tags to distinct VCIs is the possibility of wildcards on receive operations. Since HYPRE does not use wildcard communication, however, we can use the new Info hints in the draft MPI-4.0 standard to convey this information to the MPI library so that the library can exploit the parallelism information in the tags while mapping operations to VCIs. We create a new communicator that requests as many VCIs as there are threads, and we inform the MPI library, through the hints described in Section 5.3, which bits of the tag encode information about logically parallel communication. With a one-to-one mapping between thread IDs and the underlying VCIs, this approach uses VCIs in a manner similar to that of user-visible endpoints.

We implemented the different mechanisms of expressing logically parallel communication in the HYPRE library [54]. Figure 13 compares the performances of these different mechanisms on 8 KNL nodes with 1 patch per core. Communicators perform better than the original (all threads use the same communicator) version of MPI+threads but do not perform the best because of conflicts on VCIs that are

mapped to multiple communicators. Tags (with hints), on the other hand, efficiently use VCIs and perform as well as the upper bound set by user-visible endpoints. This upper bound is faster than not only the original version of MPI+threads, but also MPI everywhere. The latter is due to the lack of intranode MPI communication which results in over 70% lesser volume of MPI communication in MPI+threads as we can see in Figure 14.

Using the best-performing mechanism of expressing logical communication parallelism (e.g. tags with hints), we evaluate the impact of dissolving HYPRE's communication bottleneck on Uintah's overall performance [55]. Using 1 patch per core, we vary the domain size from 8^3 patches (8 nodes) to 20^3 patches (125 nodes). Figure 15 shows that MPI+threads with logically parallel communication performs, on average, 2.23× faster than the original version and 1.82× faster than MPI everywhere (when MPI everywhere is able to run). Thus, exposing logically parallel communication enables Uintah to achieve the best of both worlds: high scalability and high productivity.

6.2 WOMBAT

In Section 3.2, we learned that the parallel issue of RMA operations is a major bottleneck for the MPI+threads version of WOMBAT. Since these parallel operations are independent, one can eliminate the communication bottleneck by explicitly exposing the communication independence to a fast MPI+threads library.

User-visible endpoints. To expose logical parallelism with user-visible endpoints, we create as many endpoints as there are threads. Each thread uses a dedicated endpoint to issue its RMA operations. In this way, we can expose maximal communication independence between threads.

Windows. With existing MPI mechanisms, we can express communication parallelism by creating as many windows as there are threads. Similar to user-visible endpoints, each thread uses its own window to issue RMA operations.

We implemented the different mechanisms of exposing communication independence on the publicly available version of WOMBAT [56], and evaluated their performances on 27 nodes of the HPC3 cluster (Skylake UCX/IB) with a workload at the strong-scaling limit (1 patch per core). Since the nodes consist of 2 sockets with 20 cores each, we used a process per socket with 20 threads per process for MPI+threads. We first studied the communication performance differences of the different mechanisms of exposing logical parallelism. Figure 16 shows that exposing logical communication parallelism in MPI+threads reduces the time spent in MPI communication by over 90%, whether through user-visible endpoints or windows. Once the communication bottleneck in MPI+threads is eliminated, the communication time of MPI+threads is less than that in MPI everywhere because of the lesser amount of intranode MPI communication in MPI+threads. Figure 17 shows that the number of `MPI_Put` operations in MPI+threads is 50% lesser than that in MPI everywhere. Similarly, we measured the `MPI_Get` operations to be 36.15% lesser in MPI+threads.

Figure 18 shows that the reduction in MPI time with logically parallel communication boosts the overall scientific throughput of MPI+threads, especially for smaller

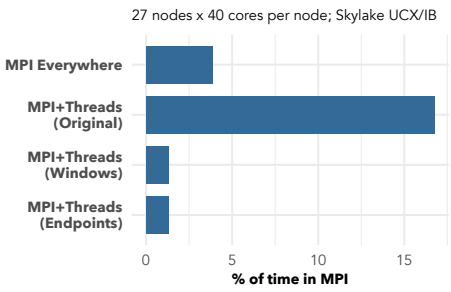


Fig. 16: Different mechanisms of exposing logical communication parallelism for WOMBAT (1 32^3 patch per core).

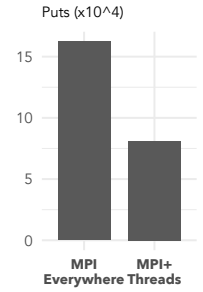


Fig. 17: MPI Puts per node.

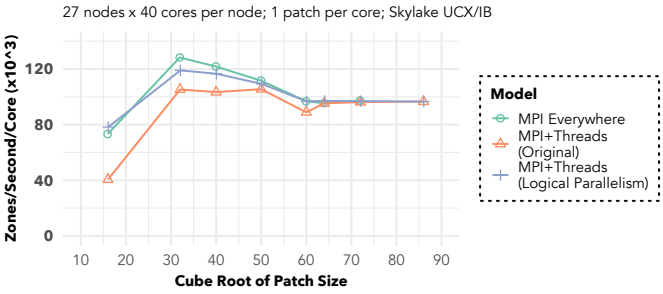


Fig. 18: Scientific throughput of WOMBAT with logically parallel communication.

patch sizes. Where MPI everywhere is able to run, however, MPI+threads with exposed communication parallelism is still slower. For example, for a patch size of 32^3, MPI+threads with logical communication parallelism performs 13.09% faster than the original version, but performs 7.21% slower than MPI everywhere. This analysis indicates that other challenges posed by the MPI+threads programming model remain to be addressed in WOMBAT. Even though WOMBAT maintains a single OpenMP parallel region, the current structure of the code requires bulk synchronization between threads on a node at certain points in the simulation. For example, when the patch size is 32^3, we measured the average time spent in executing and waiting in OpenMP barriers to be 5.07% of the total time per iteration. Removing the barriers is not a trivial task and requires restructuring the code such that each thread can operate asynchronously throughout the single OpenMP parallel region. Once the next iteration of WOMBAT addresses the core issue of exposing logical communication parallelism in addition to the other challenges of MPI+threads programming, it can achieve a higher scientific throughput than it currently does with MPI+threads while reaping the scaling benefits that its MPI+threads version exhibits over MPI everywhere [40].

6.3 Legion

In Section 3.3, we learned that the single polling thread that processes incoming events from other nodes is the theoretical bottleneck in Legion's communication performance. Without exposed communication parallelism, the polling

thread contends with the task threads even though their operations are logically independent. The ideal situation would be one where the polling thread is always available to receive events from the task threads of remote nodes. We can achieve this ideal by exposing the logical communication parallelism to a fast MPI+threads library.

User-visible endpoints. With the flexible interface of user-visible endpoints, we can fully expose communication independence by using distinct endpoints for the polling and task threads. Since each endpoint is directly addressable, the task threads can issue their operations on distinct local endpoints (e.g., based on their thread IDs) and target the remote endpoint of the polling thread.

Communicators. On the other hand, as we learned in Section 3.3, we cannot achieve the ideal exposure of logical communication parallelism using the existing MPI standard because of the matching semantics of communicators. The communicator-based approach, however, is still better than the original version. In addition to the operations from task threads being independent, the likelihood of the polling thread contending with a task thread is less with multiple communicators. When all threads use `MPI_COMM_WORLD`, the operations of the polling thread face contention if any of the task threads are simultaneously issuing an operation. With multiple communicators, however, the polling thread's contention during the processing of events on a communicator is dependent only on a single task thread—the one issuing operations on that communicator.

Tags with hints. To eliminate any possibility of contention with the polling thread using existing MPI mechanisms, we can leverage the flexibility in Realm's communication requirements and express communication parallelism through tags. Even though the polling thread uses wildcards to process an event from any node, the order of matching does not matter. In other words, Realm does not rely on MPI's non-overtaking order. The draft MPI-4.0 standard allows us to convey this information to the MPI library through Info hints. With this information conveyed, we can expose the communication parallelism between task threads by encoding, for example, their thread IDs into the tags of the MPI operations. More important, if we inform the MPI library (through hints) that a single polling thread issues only receive operations and that the task threads issue only send operations, the MPI library can funnel all receive operations through a dedicated VCI that is separate from the multiple VCIs used for the send operations from different task threads. The MPI library would ensure that all send operations target the dedicated receiving VCI on the the target node. In this way, we can achieve maximum communication independence similarly to user-visible endpoints with existing MPI mechanisms.

We implemented the different mechanisms of expressing communication parallelism in Legion's MPI backend [57], and evaluated the Circuit simulation application [41] on Bebop's Broadwell OFI/OPA cluster. Figure 19 shows the time taken by the polling thread on the critical path to process incoming events for the different mechanisms of exposing communication parallelism. This time does not include the time for polling; rather, it includes the time taken after a successful poll (receiving the payload, executing the event handler, posting a replacement receive, etc.). We observe that all mechanisms enable the polling thread to process events faster than the original version, which does not expose any communication independence. Among the existing MPI mechanisms, however, tags outperform communicators because, with tags, the polling thread does not contend with any other thread. In this way, tags with hints achieve the upper bound set by user-visible endpoints. Although our evaluation is centered around the theoretical bottleneck: a single polling thread, we note that exposing logically parallel communication reduces the time taken by the task threads to issue operations as well. The effects of the different mechanisms on the task threads is similar to those portrayed in Figure 19, that is, communicators perform better than the original version but are not as good as endpoints or tags with hints.

†. All Legion processes spawn a polling thread and hence there is no MPI everywhere version.

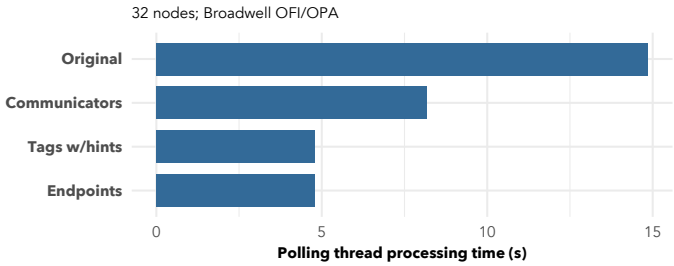


Fig. 19: Polling thread processing on Circuit's critical path†.

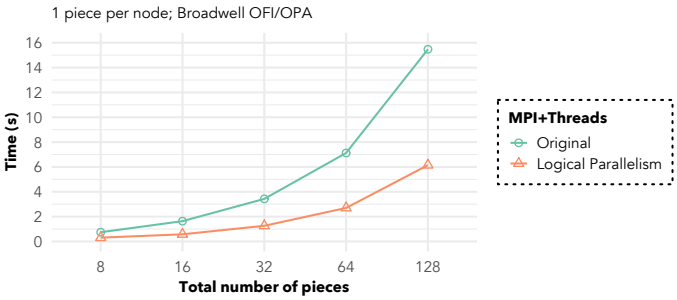


Fig. 20: Circuit simulation performance with logically parallel communication.

Figure 20 shows the overall performance of the Circuit simulation with workloads run at the strong-scaling limit (1 piece per node). Once we are able to expose all of the available logical communication parallelism by funneling the operations of Realm's polling and task threads through distinct communication channels, Figure 20 shows that we can improve the Circuit simulation performance by 2.62× on average. Not only does logically parallel communication enable the single polling thread design to achieve the best performance (by eliminating thread contention), but it also enables the exploration of backend designs with multiple polling threads. Without it, multiple polling threads would, in fact, perform slower than a single polling thread because of higher contention between threads. Exploring the benefits of multiple polling threads warrants an independent study that is specific to Legion since the exploration requires intrusive changes that also need to respect Legion's semantics. Such a study is out of the scope of this paper.

7 Discussion

A programming perspective. Our performance evaluations on applications show that the hints in MPI-4.0 enable an application to perform as well as the upper bound set by user-visible endpoints. While setting these hints in applications is trivial, aligning them with the behavior of the application can be challenging. To enable fast multithreaded communication using hints, one needs to understand the MPI semantics that an application relies on for correct behavior. The consequence of setting hints that disable needed semantics is erroneous application behavior. Thus, the user requires expertise in both the application's communication pattern and the semantics of MPI communication. Aligning hints to an application's behavior is currently a manual process. Recent research efforts, however, are attempting to automate this process through, for example, a static analysis of the program [58]. On the other end, user-visible endpoints have their own programming challenges. In addition to learning the new concept of endpoints, users need to use new APIs to create endpoints. However, given that endpoints take on the semantics of a rank, an object that users are already familiar with, we find endpoints to be more intuitive to use than hints especially for irregular communication patterns such as that of Legion (see Section 6.3).

Comparison against new alternate threading models. The draft MPI-4.0 standard features a new type of operation—Partitioned Communication—which allows multiple *actors* (e.g., threads) to contribute distinct data partitions of a single message in parallel [59], [60]. However, since threads share a single message (i.e., a single request), only a single thread can complete a partitioned communication operation. In other words, although each thread can issue its partition of a message as soon as the partition is ready, all threads need to synchronize before issuing their partition of the next message. In all the application case studies in this paper, the threads operate completely independently of each other. Partitioned communication would expose more parallelism than the state of the art (which exposes none), but, unlike existing MPI mechanisms or endpoints, it cannot expose all of the communication independence between threads. Because of this theoretical limitation, we do not evaluate the partitioned communication model in our study. Solutions to overcome the inherent synchronization of partitioned communication need to be explored. A comparison between the mechanisms in this paper and partitioned communication would make a worthy future study once the implementation of the new interface becomes available.

8 Related Work

Since MPI libraries have only recently (starting in 2019) been optimized to utilize network parallelism, the number of MPI+threads applications that have worked on alleviating their communication bottleneck is limited. We discuss how recent efforts both by MPI libraries and by applications compare to our work with respect to removing the decade-long communication bottleneck in MPI+threads.

8.1 MPI Libraries

Initial efforts by MPI libraries to improve MPI+threads communication performance focused primarily on mitigating lock contention [52], [53], [61]. Recent efforts, however, address the root of the problem: underutilization of network parallelism. Similar to VCIs in MPICH, Open MPI introduces Communication Resource Instances (CRIs) to establish parallel communication channels [22], [62]; and Intel MPI, since its 2019 release, has utilized Intel Omni-Path's network parallelism through its multiple endpoints support [23]. Both efforts, however, suffer from shortcomings in both performance and correctness.

Open MPI's CRIs, for instance, are not able to achieve scaling communication performance for point-to-point operations even with user-exposed logically parallel communication. Furthermore, CRIs do not take into consideration the notion of shared progress between threads, hence sacrificing correctness. Intel MPI's multiple-endpoints support is for a nonstandard threading level—`MPI_THREAD_SPLIT`—which does not cover all cases possible in the `MPI_THREAD_MULTIPLE` threading level. In contrast to these works, our MPI implementation with VCIs does not sacrifice correctness for performance and is able to achieve scaling communication performance for both point-to-point and RMA operations.

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 address the communication bottleneck present in applications through logically parallel communication. Comparing against other MPI libraries is orthogonal to the goal of this work, but such a comparison would make a worthwhile future study.

8.2 Application Case Studies

Electronic structure codes. The Vienna Ab initio Simulation Package (VASP) is a legacy code that is widely used for atomic-scale modeling during the development of new materials. Wende et al. [63] recently ported VASP's MPI-only codebase to MPI+OpenMP to run on the new and upcoming many-core processor architectures. Noting the bottleneck in the collective communication of their MPI+OpenMP version, they expose logically parallel communication by partitioning the large allreduce collective between threads and using a separate communicator per thread. With Intel MPI's multiple-endpoint support, they are able to achieve performance up to 1.27× faster than a version of the code without logical communication parallelism. Their work, however, is restricted to semi-hybrid configurations since they observed slower performance with fewer MPI ranks per node, preventing electronic structure codes from achieving the maximum potential of MPI+threads. This observation indicates that other bottlenecks, such as multiple parallel-for loops, remain to be addressed in legacy electronic structure codes.

Miniapp study. Boyle et al. [64] studied the impact of using Intel MPI's multiple endpoint support on the communication patterns of stencil (point-to-point operations in the Grid library [65]) and machine learning codes (collective operation in the baidu-allreduce library [66]). For both patterns, they demonstrated that a single MPI rank is able to achieve high network utilization through the use of multiple

communicators in multithreaded MPI communication. Like our work, they focus on workloads at the strong-scaling limit, but unlike our work, they do not compare against flat MPI versions of the miniapps. More important, our work focuses on the end-to-end runtime of applications rather than miniapps.

9 Concluding Remarks

In this work, we eliminate the multithreaded communication performance bottleneck present today in MPI+threads applications. We do so by exposing the communication independence in an application's multithreaded communication through *logical communication parallelism*. We leverage the new opportunities of using existing MPI mechanisms that will soon be available in MPI-4.0, the next iteration of the MPI standard. The mechanisms of MPI-4.0 allow applications to expose communication independence as well as user-visible endpoints, the upper bound in exposing logically parallel communication. A communication-intensive application realizes a significant boost (over 2×) in its overall performance when a fast MPI+threads library maps the application's logically parallel communication to the underlying network parallelism. We emphasize, however, that the benefits of exposing logical communication parallelism are most visible when the application has maximized the independence between threads for both computation and communication. We encourage domain experts to address the other challenges of MPI+threads programming in their codes so that their applications can achieve both higher scalability and higher productivity over the traditional MPI everywhere model through logical communication parallelism.

Acknowledgments

We thank Peter Mendygral from HPE Cray for his help and guidance for our evaluation with WOMBAT. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory (ANL), on HPC3, a high-performance community computing cluster operated by the Research Cyberinfrastructure Center at the University of California, Irvine, and by the Joint Laboratory for System Evaluation at ANL. We thank the continuous feedback from the members of the PMRS group at ANL, and we thank Gail Pieper from ANL for her timely edits on this paper. This work is supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and the National Science Foundation under the award number 1750549.

References

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

[2] "Intel and AMD face an Arm'ed onslaught from this 96-core CPU monster," https://www.techradar.com/news/intel-and-amd-face-an-armed-onslaught-from-a-96-core-cpu-monster.

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

[4] 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*. IEEE, 2009, pp. 427–436.

[5] 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*. Springer, 2014, pp. 3–27.

[6] 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.

[7] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[8] D. Karlbom, "A performance evaluation of mpi shared memory programming," 2016.

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

[10] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S101–S122, 2016.

[11] 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*, vol. 222, pp. 351–373, 2018.

[12] 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*, vol. 37, no. 9, pp. 562–575, 2011.

[13] E. Higgins, M. Probert, P. Hasnip, K. Refson, and I. Bush, "Hybrid openmp and mpi within the castep code," ARCHER eCSE Technical Report, Tech. Rep., 2015.

[14] 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*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 109120. [Online]. Available: https://doi.org/10.1145/3307681.3325402

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

[16] 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*. Springer, 2015, pp. 451–463.

[17] K. G. Felker, A. R. Siegel, K. S. Smith, P. K. Romano, and B. Forget, "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, 2014, p. 04207.

[18] H. Wang and A. Chandramowlishwaran, "Multi-criteria partitioning of multi-block structured grids," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 261–271.

[19] ——, "Pencil: a pipelined algorithm for distributed stencils," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16.

[20] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypres multigrid solvers to 100,000 cores," in *High-Performance Scientific Computing*. Springer, 2012, pp. 261–279.

[21] 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*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392773

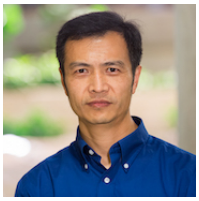
[22] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019.

[23] "Intel® MPI Multiple Endpoints Sup-

port," https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support.
[24] S. Sridharan, J. Dinan, and D. D. Kalamkar, "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, 2014, pp. 487–498.
[25] 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*, vol. 28, no. 4, pp. 390–405, 2014.
[26] "MPI Endpoints," https://github.com/mpi-forum/mpi-issues/issues/56.
[27] 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*. ACM, 2013, pp. 13–18.
[28] 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*. ACM, 2017, p. 62.
[29] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "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, 2015, pp. 34–39.
[30] P. Shamis *et al.*, "UCX: an open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performane Interconnects*. IEEE, 2015, pp. 40–43.
[31] M. Berzins, "Status of release of the Uintah computational framework," *Scientific Computing and Imaging Institute, Tech. Rep. UUSCI-2012-001*, 2012.
[32] A. Humphrey, T. Harman, M. Berzins, and P. Smith, "A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing," in *International Conference on High Performance Computing*. Springer, 2015, pp. 212–230.
[33] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on Computational Science*. Springer, 2002, pp. 632–641.
[34] Q. Meng and M. Berzins, "Scalable large-scale fluid–structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1388–1407, 2014.
[35] A. Humphrey and M. Berzins, "An Evaluation of An Asynchronous Task Based Dataflow Approach For Uintah," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2019, pp. 652–657.
[36] J. K. Holmen, B. Peterson, and M. Berzins, "An approach for indirectly adopting a performance portability layer in large legacy codes," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 36–49.
[37] Q. Meng, M. Berzins, and J. Schmidt, "Using hybrid parallelism to improve memory use in the Uintah framework," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, 2011, pp. 1–8.
[38] 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*. Springer, 2020, pp. 175–190.
[39] "WOMBAT & CosmoPlasma," https://wombatcode.org/.
[40] 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*, vol. 228, no. 2, p. 23, 2017.
[41] 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*. IEEE, 2012, pp. 1–11.
[42] 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*, 2014, pp. 263–276.
[43] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, 2007, pp. 24–32.
[44] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 495–514, 2013.
[45] D. Holmes, "Introducing Endpoints into the EMPI4Re MPI library."
[46] "MPI-4.0 Draft Report," https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf.
[47] "Multi-VCI MPICH with OFI," https://github.com/rzambre/mpich/tree/mpi-endpoints-ofi.
[48] "Multi-VCI MPICH with UCX," https://github.com/rzambre/mpich/commits/vci-pool-with-ucx.
[49] "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.
[50] 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)*. IEEE, 2018, pp. 803–812.
[51] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2008, pp. 120–129.
[52] P. Balaji *et al.*, "Fine-grained multithreading support for hybrid threaded mpi programming," *The International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 49–57, 2010.
[53] A. Amer, C. Archer, M. Blocksome, C. Cao, M. Chuvelev, H. Fujita, M. Garzaran, Y. Guo, J. R. Hammond, S. Iwasaki *et al.*, "Software combining to mitigate multithreaded MPI contention," in *Proceedings of the ACM International Conference on Supercomputing*. ACM, 2019, pp. 367–379.
[54] "Hypre with logically parallel communication," https://github.com/damu1000/hypre_ep.
[55] "Uintah using hypre with logically parallel communication," https://github.com/damu1000/Uintah.
[56] "WOMBAT with logically parallel communication," https://bitbucket.org/rzambre/wombat-public/commits/branch/master.
[57] "Legion's MPI backend with logically parallel communication," https://github.com/rzambre/legion/commits/vci_parallelism.
[58] T. Jammer, C. Iwainsky, and C. Bischof, "Automatic detection of mpi assertions," in *International Conference on High Performance Computing*. Springer, 2020, pp. 34–42.
[59] R. Grant, A. Skjellum, and P. V. Bangalore, "Lightweight threading with mpi using persistent communications semantics." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.
[60] 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*. Springer, 2019, pp. 330–350.
[61] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+ threads: Runtime contention and remedies," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 239–248, 2015.
[62] A. Gopalakrishnan, M. A. Cabral, J. P. Erwin, and R. B. Ganapathi, "Improved MPI Multi-Threaded Performance using OFI Scalable Endpoints," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2019, pp. 36–39.
[63] F. Wende, M. Marsman, J. Kim, F. Vasilev, Z. Zhao, and T. Steinke, "OpenMP in VASP: Threading and SIMD," *International Journal of Quantum Chemistry*, vol. 119, no. 12, p. e25851, 2019.
[64] 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.
[65] "Multi-communicator benchmark using the Grid library," https://github.com/paboyle/Grid/blob/develop/benchmarks/Benchmark_comms.cc.
[66] "baidu-allreduce C++ library modified to use mulitple communicators," https://github.com/paboyle/baidu-allreduce.

Rohit Zambre is a Ph.D. candidate at the University of California, Irvine, where he also received his M.S. degree in 2017. He received his B.S. degree from Iowa State University in 2011. During his Ph.D., he interned at Arm Research, and held a Visiting Student appointment at Argonne National Laboratory. His research interests lie broadly in supercomputing networking technologies, and programming models for multithreaded environments.

Damodar Sahasrabudhe is a Ph.D. candidate at the University of Utah. He worked in the software development industry for nine years in various roles including Software Engineer and Technical Architect before joining the Ph.D. program at Utah in 2015. His research interests include massively parallel solutions for the complex scientific problems, development of efficient programming models for GPUs and/or multithreaded environments, performance portability and optimizations using GPUs and multithreading, and asynchronous many task runtime systems.



Hui Zhou is a Software Engineer at Argonne National Laboratory. He is the principal software developer for the MPICH project. Before he joined Argonne in 2018, he worked as a computational physicist at National Institute of Standards and Technology, where he developed and maintained an in-house finite-difference-time-domain simulation software and researched in the area of hybrid metrology. He is still very interested in physics and science in general, but his current focus is to explore how the scientific software can be developed in a scalable and maintainable way.

Martin Berzins is a Professor of Computer Science in the School of Computing and the SCI Institute at the University of Utah. He is an Adjunct Professor of Mathematics in Utah and a Visiting Professor at the University of Leeds. He moved to Utah in 2003 from the University of Leeds in the UK where he went from being a student to eventually being the Research Dean of Engineering. He has worked in the fields of mathematical software, numerical analysis, and parallel computing for challenging problems in science and engineering. His recent research has been concerned with portable parallel computing at extreme scales on heterogeneous architectures.

Aparna Chandramowlishwaran is an Associate Professor at the University of California, Irvine, in the Department of Electrical Engineering and Computer Science. She received her Ph.D. in Computational Science and Engineering from Georgia Tech in 2013 and was a research scientist at MIT prior to joining UCI as an Assistant Professor in 2015. Her research lab—HPC Forge—aims at advancing computational science using high-performance computing and artificial intelligence. She currently serves as the associate editor of the ACM Transactions on Parallel Computing.

Pavan Balaji holds appointments as a Computer Scientist and Group Lead at the Argonne National Laboratory, where he leads two groups: Programming Models and Runtime Systems and Future Architectures for AI. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, deep learning, and big-data sciences. He is a senior member of the IEEE and a professional member of the ACM.