UNIVERSITY OF CALIFORNIA,
IRVINE

Exascalable Communication for Modern Supercomputing

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Rohit Zambre

Dissertation Committee:
Associate Professor Aparna Chandramowlishwaran, Chair
Professor Athina Markopoulou
Assistant Professor Ardalan Amiri Sani

2020

DEDICATION

To my parents, Shahaji and Ujwala Zambre
for their immense sacrifices, and to serendipity.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGMENTS

My Ph.D. journey consists of many serendipitous events that have impacted both my academic and personal lives in significant ways. I am extremely thankful for their occurences.

First and foremost, I would like to thank Aparna for beleiving in me and taking me on as her student. I am extremely grateful to her technical, writing, and personal advices throughout my Ph.D. Her lessons have taught me to never lose sight of the bigger picture both in research and in life. Her constant motivation helped me push through the toughest of times. Thank you, Aparna, for your guidance in navigating my Ph.D. journey and always encouraging me to strive for the best.

I thank Pavan Balaji from Argonne National Laboratory (ANL) for supporting my Ph.D. and providing me with the opportunity to work on an important problem that allowed me to experience research first-hand at a national-lab setting. I thank him for pushing the boundaries of my capabilities and teaching me how to conduct incremental and in-depth research. His technical advices and detailed help on papers have been invaluable. Thank you, Pavan.

I thank Pasha (Pavel Shamis) from Arm Research for providing me with a unique opportunity to play with super expensive PCIe analyzers and experience research at an industry-research environment. His playful enthusiasm about network technologies and emphasis on work-life balance was refreshing to witness. Thank you, Pasha, for the very interesting and rewarding opportunity.

I thank Athina Markopoulou and Ardalan Amiri Sani for always being there to support me, and for taking the time to serve on both my candidacy and thesis committees.

I am extremely grateful to have had the best of labmates both at UC Irvine and at ANL. Shintaro at ANL played an integral role in my learning phase and I am thankful for the countless discussions, lessons, and pointers, that he patiently provided me with. Thanks to Kaiming for the fruitful research discussions and for patiently listening to me during tough times. Giuseppe, Kavitha, and Sridutt at ANL have been both great mentors and friends and I thank them for their support and fun times. I have made very fond memories with my HPC Forge labmates—Laleh, Bahareh, Behnam, Shu-Mei, Octavi, and Hengjie—both in and out of lab. From paintball to skiing to just goofing around in the lab and at conferences, I thank them for all the invaluable memories.

My friends at UCI have kept me refreshed throughout my Ph.D. Thanks to Ronit for showing me around Orange County and Los Angeles, for always introducing me to great music, and for all the phone calls and voicemails. Thanks to Nikhil for the good times in Irvine, for showing me around Austin, and for teaching me the ropes of negotiating a job offer. Thanks to Primal and Anirudh for being my first Ph.D. friends at UCI and for introducing me into their circle of friends—Dhrub, Nitin, Roberto, and Pele. The many hiking trips we took together have been key highlights of my time at UCI. Thanks to Marwen for being a great friend throughout the Ph.D. and for graciously hosting me with top-notch breakfasts during my brief stays in Irvine. Special thanks to Primal for helping me get through the beginning phases of the pandemic and for taking me grocery shopping when I didn't have a car. Special thanks also to Tanya, Prabhat, and Aishwarya for the pandemic movie nights and memorable birthday celebrations.

I am thankful to have met my friends Chanpreet, Preet, and Rohan in Chicago who not only showed me around, but also helped me get through the windy city's long winters. The road trips and parties were excellent stress relievers. Special thanks to them for introducing me to Jeni's ice cream, my go-to for any celebration now.

Through my internships, I have made long-lasting friendships and that have supported me continuously throughout the Ph.D. Thanks to Anthony (whom I met at Arm) for always coming out to support me be it at a virtual recorded presentation or a live defense. Thanks to Martin (whom I met at Argonne) for his continued support from France.

A big thanks to Lucy for sticking by my side through all the stressful paper deadlines, and for helping me move across the country on various occasions. Without her help, sympathy and support, my Ph.D. journey wouldn't have had as many happy moments as it did. Thank you, Lu. Special thanks to Lucy's parents—John and Jackie—for hosting me at their home and giving me a comfortable place to stay as I wrote this dissertation.

Thank you Mama and Baba for your immense sacrifices over the years, and thank you for supporting my freedom in pursuing a Ph.D. Special thanks to my little sister, Mrunal, for her support and for regularly paying me visits no matter where I was located.

VITA

Rohit Zambre

EDUCATION

Doctor of Philosophy in Computer Engineering	**2020**
University of California, Irvine	*Irvine, CA*
Master of Science in Computer Engineering	**2017**
University of California, Irvine	*Irvine, CA*
Bachelor of Science in Electrical Engineering	**2015**
Iowa State University	*Ames, IA*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2015–2020**
University of California, Irvine	*Irvine, CA*
Visiting Student	**2017–2020**
Argonne National Laboratory	*Lemont, IL*
Research Intern	**Summer 2018**
Arm Research	*Austin, TX*
Research Aide	**Summer 2017**
Argonne National Laboratory	*Lemont, IL*
Research Assistant	**Summer 2016**
Mozilla Research	*Irvine, CA*

TEACHING EXPERIENCE

Graduate Teaching Assistant	**Fall 2016**
University of California, Irvine	*Irvine, CA*
Peer Mentor	**2012–2015**
Iowa State University	*Ames, IA*
Undergraduate Teaching Assistant	**Fall 2013**
Iowa State University	*Ames, IA*

ENGINEERING EXPERIENCE

Hackathon Participant **January 2020**
Intel Corporation *Austin, TX*

Intern SSD Modeling Engineer **Summer 2014**
Micron Technology, Inc. *Longmont, CO*

NAND Product Engineering Intern **Summer 2013**
Micron Technology, Inc. *Boise, ID*

REFEREED CONFERENCE PUBLICATIONS

How I Learned To Stop Worrying about User-Visible Endpoints and Love MPI **June 2020**
34th ACM International Conference on Supercomputing (ICS)

Breaking Band: A Breakdown of High-Performance Communication **August 2019**
48th ACM International Conference on Parallel Processing (ICPP)

Scalable Communication Endpoints for MPI+Threads Applications **December 2018**
24th IEEE International Conference on Parallel and Distributed Systems (ICPADS)

Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo. **December 2016**
23rd IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)

ABSTRACT OF THE DISSERTATION

Exascalable Communication for Modern Supercomputing

By

Rohit Zambre

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2020

Associate Professor Aparna Chandramowlishwaran, Chair

Supercomputing applications rely on strong scaling to achieve faster results on a larger number of processing units. But, at the strong-scaling limit, where communication is a relatively large portion of an application's runtime, today's state-of-the-art hybrid MPI+threads applications perform slower than their traditional MPI everywhere counterparts. This slowdown is primarily due to the supercomputing community's outdated view: the network is a single device. NICs of modern interconnects feature multiple network hardware contexts. These parallel interfaces into the network are not utilized in MPI+threads applications today because MPI libraries still use conservative approaches to maintain MPI's ordering constraints. MPI libraries do so because domain scientists today do not do a good job exposing logically parallel communication in their multithreaded MPI communication even though the existing MPI standard provides them with opportunities to do so. Only when domain scientists and MPI developers take a step forward together can we eliminate the communication bottleneck in MPI+threads applications.

This dissertation eliminates the communication bottleneck by bridging the two ends of the HPC stack—MPI library developers and domain experts—that typically do not talk to each other directly. Through collaborations with system researchers and MPI library developers, we develop a fast MPI+threads library capable of achieving scaling communication throughput similar to that of MPI everywhere and make high-speed multithreaded communication a reality. Through collaborations with domain scientists, we use various designs to expose logically parallel communication to the fast MPI+threads library on exemplar applications targeted to run on the upcoming exascale systems. Our conversations with the end-users—the domain experts—educate us on the usability aspects of the various designs. Hence, in addition to the performance comparisons of the various designs, we discuss the strengths and limitations of the different designs and provide our design recommendation for the supercomputing community. Through such collaborations on both ends of the HPC stack, we unlock the true potential of the MPI+threads programming model. Prominent modern applications and computational frameworks, such as Uintah, WOMBAT, and Legion, now perform significantly faster (up to 2x) at the strong-scaling limit.

Chapter 1

Introduction

Multiple national-level efforts across the globe are striving to reach the next frontier in computing: exascale computing. In the pursuit of achieving an exaflop, that is, 10^{18} floating point operations per second, the architectures of supercomputers have critically evolved over the last decade, jeopardizing the traditional models of programming them. Even with modern programming models, however, domain scientists rely on the efficient use of a larger number of parallel processing units to achieve faster scientific productivity (strong scaling) [76, 47]. At the limits of strong scaling, the workload per processing unit is small and the communication between the units occupies a significant portion of an application's runtime. We can witness this behavior in both the scientific computing and AI domains. Scientific computing typically consists of stencil-style workloads with neighborhood communication that takes up a large fraction of the runtime at scale [102, 103, 91], and communicating gradients in deep learning is a major bottleneck for distributed DNN training at scale [104, 65]. Hence, as we inch towards exascale computing, it is critical that applications are able to achieve high parallel efficiencies with scalable communication performances using modern programming models.

1.1 Background and Architecture Trends

The Message Passing Interface (MPI) has been the de facto standard for programming supercomputers and it is typically used in a SIMD fashion by launching a process per core. This approach is commonly referred to as MPI everywhere since it enables the domain scientist to transparently utilize both the core-level and node-level parallelism on large-scale distributed systems. The traditional MPI everywhere programming model has served the community well for decades, but it is no longer a good fit for modern processors, and it is not enough to program modern supercomputers. Two architecture trends have primarily imperiled MPI everywhere.

Trend 1. As the serial throughput of a CPU has stagnated, vendors have increased the number of cores per processor over the last decade to achieve high aggregate computational throughput. This trend remains to continue on the upcoming exascale supercomputers (e.g., 64 cores on the Frontier exascale machine of the US) [6, 97]. An important aspect of this trend is that the processor's other resources, such as memory, have not increased proportionally to the increase in the number of cores. We can witness this trend even on the world's fastest supercomputer (as of November 2020), Fugaku, where the ratio of memory to core is less than one (48 cores with 32 GB RAM per node). The MPI everywhere model falters on today's many-core processors because of its high memory requirements [86]. In domain decomposition scenarios, for example, the halo regions of processes contain duplicated data, which increases with the increase in the domain's dimensionality. More important, the static split of resources with MPI everywhere on such many-core processors leads to inefficient resource usage because of resource wastage [106].

Trend 2. The second, more recent trend, has been the move towards heterogeneous computing environments through multiple accelerators per node. The Sierra and Summit supercomputers, for example, feature 4 and 6 GPUs per node, respectively. More important, all the upcoming US exascale supercomputers—El Capitan, Frontier, and Aurora—will feature multiple GPUs per node. To fully utilize the computational capability of a modern supercomputer, domain scientists must program both processors and GPUs, and hence cannot rely on using only MPI everywhere, that is, they must use a GPU programming model (e.g. CUDA) to utilize the massive parallelism within a GPU. Additionally, modern GPU-based distributed applications use MPI to utilize the GPU-level parallelism by using multiple processes to drive the multiple GPUs on the node [70, 63, 69, 28]. The MPI everywhere approach used in such distributed GPU-based applications, however, suffers from the same scaling problems resulting from trend 1.

1.2 The Problem with Modern Programming Models

To utilize the modern many-core processors in a scalable fashion, domain scientists have introduced a ″new normal″ in their programming model of choice: MPI+threads (e.g., MPI+OpenMP). In this approach, the user spawns a process per node or NUMA domain and a thread per core, enabling core-level parallelism through threads and node-level parallelism through MPI processes. The MPI+threads model is a better fit than MPI everywhere for modern processors that feature a decreased share of resources per core because it allows applications to efficiently share the processor′s resources between cores. Additionally, it has lesser memory requirements—-in the example of domain decomposition scenarios, MPI+threads eliminates the duplication of data on a node and reduces the amount of memory required by a factor of the number of threads compared to MPI everywhere. As a result, many applications that have ported their codes to use the MPI+threads programming model are able to scale to a large number of nodes on modern systems [35, 42]. Evidently, MPI+threads is a critical model to program the many-core processors of exascale machines in a scalable fashion.

Although MPI+threads helps applications to scale, in practice, the MPI+threads version of an application tends to perform slower than the application's MPI everywhere counterpart [64, 38, 55]. MPI+threads programming includes many challenges over MPI everywhere. These include mitigating synchronization overheads of the shared-memory programming model [54, 88, 61], and preventing performance-degrading memory accesses (e.g., false-sharing effects). But the most important challenge is the dismal communication performance of an MPI+threads application that is most visible at the strong-scaling limit. The MPI+threads version of the hypre solver [29], for instance, spends 2.81× more time in MPI than does its corresponding MPI everywhere version. This drastically slow multithreaded MPI performance is also the reason why GPU-based applications use multiple processes instead of threads to drive the communication for distinct GPUs. The problem lies in the multithreaded issue of MPI communication operations from a single process.

State-of-the-art MPI libraries use conservative approaches, such as a global critical section, to maintain thread safety and MPI's ordering constraints which severely limit the overall performance of MPI+threads applications. Several research efforts aim to mitigate thread contention in the MPI library through fine-grained critical sections, software combining, and dedicated communication threads [31, 27, 26, 100], but such efforts do not improve the scalability of multithreaded communication performance. The root of this limiting communication performance scalability stems from an outdated view held by the supercomputing community: the network is a single device.

Trend 3. A trend that has been overlooked by the supercomputing community is the increase in network parallelism available on a single node of a supercomputer. The network interface cards (NICs) of modern interconnects, such as Nvidia-Mellanox InfiniBand and HPE Cray Slingshot, feature multiple network hardware contexts. These contexts serve as

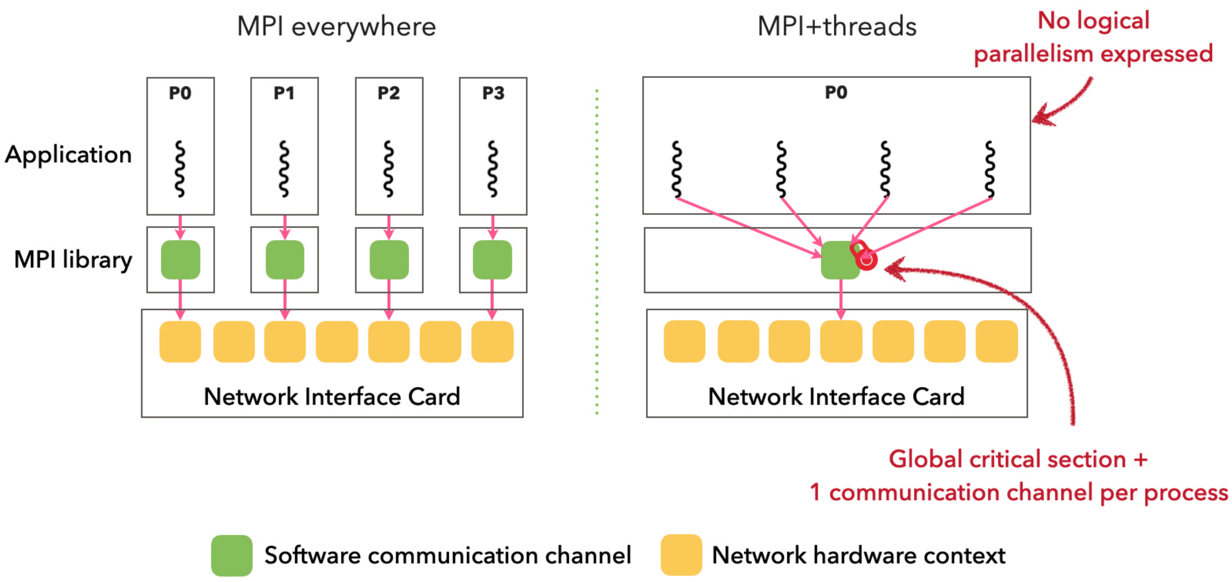


Figure 1.1: Network utilization in the state of the art.

parallel interfaces into the network from a single node. Looking forward, as we approach the physical limits of a single network link's throughput, the only way to increase the aggregate throughput from a node is through the use of multiple NICs, implying another dimension to network parallelism. The Summit supercomputer, for example, already features dual-rail Mellanox EDR InfiniBand on each of its nodes.

The outdated view does not hurt MPI everywhere applications because each MPI process transparently maps to a distinct network hardware context and utilizes the available network parallelism (see Figure 1.1). But the same outdated view hurts multithreaded MPI communication drastically—MPI libraries today map to only a single network hardware context and do not capitalize on the available network parallelism with MPI+threads. More important, domain scientists today do not do a good job exposing the independence between the MPI communication from multiple threads even though the existing MPI standard provides them with opportunities to do so.

1.3 Research Approach and Contributions

To effectively utilize the available network parallelism in MPI+threads applications, domain scientists must expose the communication independence in their MPI+threads applications through *logically parallel communication*—operations that are not subject to MPI's ordering constraints. Only once applications expose such information can the MPI library map the logically parallel communication to the underlying network parallelism. Of course, this mapping is possible only when the MPI library establishes parallel communication channels and enables efficient multithreaded access to the library's software resources. Unlike the other challenges of MPI+threads programming that can be addressed by the application developer alone, the challenge of efficient multithreaded MPI communication requires efforts from the supercomputing community as a whole, as we can see in Figure 1.2.

To achieve the goal in Figure 1.2, this dissertation bridges the two ends of the HPC stack—MPI library developers and domain experts—that typically don't talk to each other di-

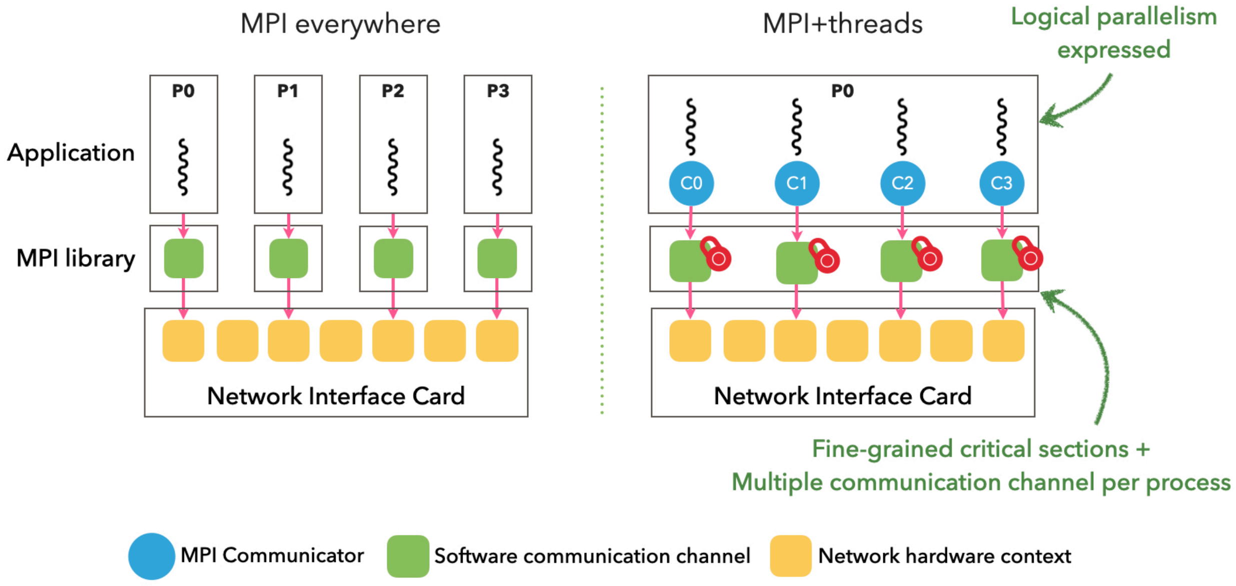


Figure 1.2: Goal of network utilization in MPI+threads matching that of MPI everywhere.

rectly. Only when both domain scientists and MPI developers take a step forward together can we observe a formidable impact on MPI+threads applications. We study each level of the HPC stack with a bottom-up approach and provide solutions for the problems at each level. This bottom-up approach was necessary since the MPI library sits in between the applications and network hardware. To accommodate the performance needs of an application, the MPI library first needs to be aware of the capabilities of modern hardware. Through collaborations with system researchers and MPI library developers, we make high-speed multithreaded MPI communication a reality. And through collaborations with application developers, we expose the applications' logically parallel communication to the fast MPI+threads library. Through both collaborations, this dissertation showcases significant boosts to performance on exemplar applications targeted to run on the upcoming exascale systems. To that end, this dissertation makes the following contributions.

1. **Analytical communication models.** Motivated by the onset of new prominent vendors, such as Arm and AMD, in HPC, we discovered that the communication performance of the new vendors is slower than that of traditional vendors such as Intel. To guide the optimization efforts of system engineers in this regard, we provide models to analyze and generate a breakdown for a system's communication performance in Chapter 2. We provide a detailed measurement methodology to measure the time spent in each component so that researchers can analyze the communication performance of any system of their interest. This work is the first of its kind for an Arm-based server.

2. **Modern network hardware capabilities.** Creating multiple communication channels for MPI+threads requires understanding the performance capabilities of modern NICs with respect to multithreaded communication. Compared to an MPI everywhere environment, an MPI+threads environment allows cores to share re-

sources efficiently. In Chapter 3, we analyze the performance impact of sharing communication resources between threads and provide a resource-sharing model that captures the tradeoff space between communication throughput and resource efficiency.

3. **A fast MPI+threads library.** Using the lessons learned from our modeling studies, we establish parallel communication streams inside the MPI library and utilize the underlying network parallelism. Our designs do not sacrifice correctness for performance. Chapter 4 details how our designs yield MPI+threads communication performance similar to that of MPI everywhere. This work is the first to achieve scaling multithreaded communication throughput for both point-to-point and RMA operations without any extensions to the MPI standard.

4. **Application case studies.** Using our fast MPI+threads library, we showcase how applications can, through logically parallel communication, achieve not only high scalability, but also high performance with MPI+threads. In Chapter 5, we compare the performance differences of the different mechanisms—user-visible endpoints and existing MPI mechanisms—of exposing logical communication parallelism for three applications from different domains: Uintah (CFD), WOMBAT (astrophysics), and Legion (data-centric programming system) applications. Exposing logical communication parallelism can boost performance by up to 2×.

5. **Listening to domain scientists.** The end-users of designs that express logically parallel communication are domain scientists. Hence, it is imperative to pay attention to the concerns of the domain scientists with respect to each design. In Chapter 6, we summarize the opinions, thoughts and concerns that we have gathered from our collaborations with several domain scientists who represent different types of MPI+threads applications.

Chapter 2

Analytical Models of Communication

Internode communication is the crux of supercomputing. Modern supercomputers feature new vendors such as Arm, AMD, and Nvidia in addition to the traditional players such as Intel and IBM. How do the communication performances of non-traditional vendors fare? Figure 2.1 shows a performance comparison between three server-class CPUs from three different vendors for a communication-intensive microbenchmark on the same Mellanox InfiniBand network. An Intel core injects messages into the network 2.38× faster than an AMD core and 1.64× faster than an Arm-based core.

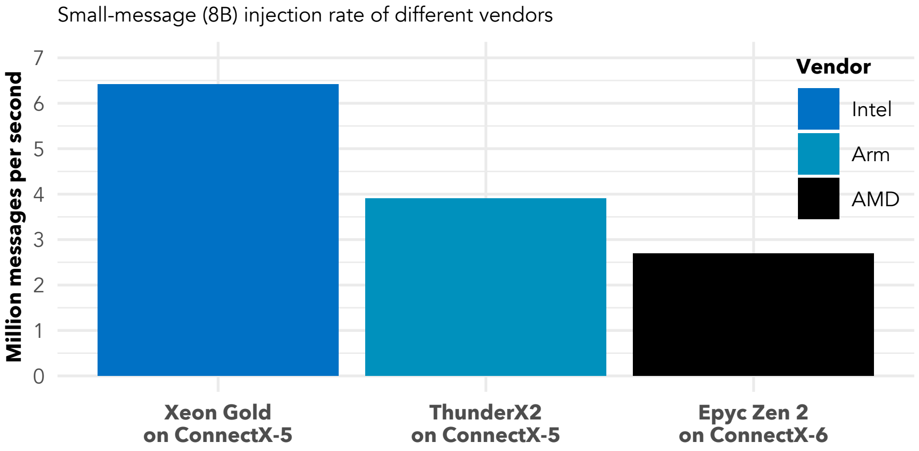


Figure 2.1: Communication performance comparison between HPC vendors.

While non-traditional vendors are better than traditional vendors at other aspects, such as memory bandwidth, they are not at par with existing machines with respect to communication performance. Finding the causes of slowdowns in communication performance is not an easy task since the critical path of communication involves all components of a system: CPU, I/O, and network. Blindly optimizing each component is impractical considering the technical challenges associated with each and the wide variety of use cases. How should system engineers then organize their optimization efforts? A detailed breakdown showing the contribution of each component to the overall communication performance of a system would provide system researchers and engineers with a holistic picture and guide their optimization efforts.

To break down high-performance communication, we study and develop analytical models for the injection overhead and end-to-end latency of a system. These models detail the time spent in all components of a system. Specifically, we focus on the communication performance of small messages because at the limits of strong scaling lies fine-grained communication of small messages. Moreover, modern networks feature optimized mechanisms for small-message communication. We use our models to break down the communication performance of a system of two Arm-based ThunderX2 servers interconnected with Mellanox InfiniBand. We choose to study an Arm-based server because the Arm architecture powers Fugaku, the fastest supercomputer in the world(as of November 2020), and we choose Mellanox InfiniBand since it occupies the highest interconnect performance share on the TOP500. This work is the first of its kind for an Arm-based server.

We learn from our study that CPU, I/O, and network components equally contribute to the communication performance of small messages; the times spent in each of the categories are on the same order of magnitude. Hence, optimizations of each category's constituents would be beneficial. This raises the question: *how much will optimizing component X improve the overall communication performance?* The answer to this question can guide the research and engineering efforts for not just the Arm-based server but also of the HPC community at large. We answer this question through a what-if analysis of the impact of optimizations on the overall communication performance.

2.1 Background

2.1.1 Communication Components

We can classify the various components involved in sending a message into one of three categories: CPU, I/O, or network fabric, as shown in Figure 2.2. Software stacks on the CPU include the Message Passing Interface (MPI) and the communication protocol processing in the underlying communication frameworks. I/O encompasses subsystems on the processor chip such as PCI Express (PCIe). Network components are the high-performance interconnects switches and physical wire. Each of these components on the critical path of communication poses an opportunity for optimization. Depending on the use case, however, the share of time spent in different components can vary. For example, the latency of sending a large message is driven by the time spent in the network components, in which case, optimizing the software stack would be a futile effort. On the other hand, the time spent in the software stack during the propagation of a small message is a considerable portion of the overall latency and, hence, optimizing the time spent in the

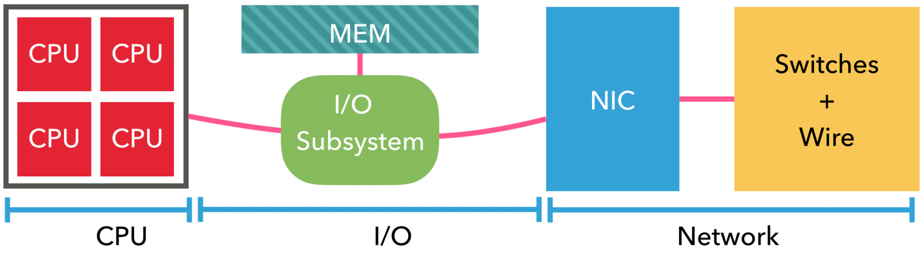


Figure 2.2: Components involved in the transmission of a message.

CPU would be beneficial. Therefore, it is important to understand where researchers and engineers should focus their optimization efforts.

2.1.2 PCIe Express

The Network Interface Cards (NICs) of modern interconnects are typically connected to the processor chip on the node as a PCI Express (PCIe) device. The main conductor of the PCIe subsystem is the Root Complex (RC). It connects the processor and memory to the PCIe fabric. The peripherals connected to the PCIe fabric are called PCIe endpoints. The PCIe protocol consists of three layers: the Transaction layer, the Data Link layer, and the Physical layer. The first, the upper-most layer, describes the type of transaction occurring. In this paper, two types of Transaction Layer Packets (TLPs) are relevant: MemoryWrite (MWr) and Memory Read (MRd). Unlike the standalone MWr TLP, the MRd TLP is coupled with a Completion with Data (CplD) transaction from the target PCIe endpoint which contains the data requested by the initiator. A MWr TLP is categorized as a *posted* transaction, meaning that the initiator of the TLP need not maintain any state regarding the transaction after issuing it. On the other hand, a MRd TLP is categorized as a *non-posted* transaction, meaning that the initiator will receive a Completion with Data (CplD) transaction from the target PCIe endpoint with the requested data. The Data Link layer ensures the successful execution of all transactions using Data Link Layer Packet (DLLP) acknowledgements (ACK/NACK) and a credit-based flow-control mechanism. An initiator can issue a transaction as long as it has enough credits for that transaction. Its credits are replenished when it receives Update Flow Control (UpdateFC) DLLPs from its neighbors. Such a flow-control mechanism allows the PCIe protocol to have multiple outstanding transactions.

2.1.3 Mechanisms of a High-Performance Interconnect

From a CPU programmer's perspective, there exists a *transmit queue* (TxQ) and a *completion queue* (CQ). The user posts their message descriptor (MD) to the transmit queue, after which they poll on the CQ to confirm the completion of the posted message. The user could also request to be notified with an interrupt regarding the completion. However, the polling approach is performance-oriented since there is no context switch to the kernel in the critical path. The actual transmission of a message over the network occurs through coordination between the processor chip and the NIC using memory mapped I/O (MMIO) and direct memory access (DMA) reads and writes. We describe these steps below using Figure 2.3.

0. The user first enqueues an MD into the TxQ. The network driver then prepares the device-specific MD that contains headers for the NIC, and a pointer to the payload.
1. Using an 8-byte atomic write to a memory-mapped location, the CPU (the network driver) notifies the NIC that a message is ready to be sent. This is called *ringing the DoorBell*. The RC executes the *DoorBell* using a MWr PCIe transaction.
2. After the *DoorBell* ring, the NIC fetches the MD using a DMA read. A MRd PCIe transaction conducts the DMA read.
3. The NIC will then fetch the payload from a registered memory region using another DMA read (another MRd TLP). Note that the virtual address has to be translated to its physical address before the NIC can perform DMA-reads.
4. Once the NIC receives the payload, it transmits the read data over the network. Upon a successful transmission, the NIC receives an acknowledgment (ACK) from the target-NIC.

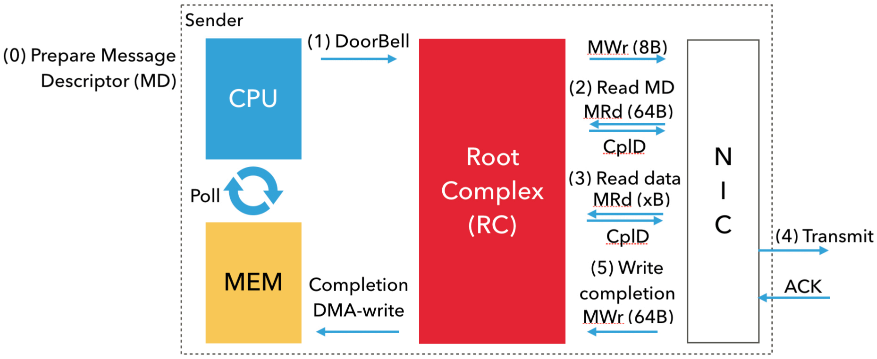


Figure 2.3: PCIe transactions and mechanisms on sender node to transmit data over wire.

5. Upon the reception of the ACK, the NIC will DMA-write (using a MWr TLP) a completion (64 bytes in Mellanox InfiniBand) to the CQ associated with the TxQ. The CPU will then poll for this completion to make progress.

In summary, the critical data path of each post entails one MMIO write, two DMA reads, and one DMA write. The DMA-reads translate to round-trip PCIe latencies which are expensive.

Network optimizations for small messages. A faster way to send a message that eliminates the PCIe round-trip latencies is *Programmed I/O (PIO)*. With PIO, the CPU copies the MD as a part of the *DoorBell*. Thus, the NIC doesn't need to DMA-read the MD. Another feature for small payloads is *inlining* which means that the payload is a part of the MD. Hence, when the NIC receives the MD, it does not need to DMA-read the payload. Typically, communication frameworks, such as UCX, combine PIO with inlining. This eliminates both the DMA-reads (steps (2) and (3)). In Mellanox InfiniBand, the PIO occurs in 64-byte chunks. Note that the CPU does more work in PIO (64-byte copy instead of an 8-byte write) and inlining (`memcpy`). However, the increase in CPU's work compared to the benefit gained from elimination of PCIe round-trip latencies is minimal.

2.2 Evaluation Setup

To measure the breakdown of time spent in communication components we use a system of two nodes, node 1 and node 2, that are connected to each other using a high-performance interconnect. Node 1 plays the role of the initiator in our following experiments. We use the CPU's timers to measure the time spent in software. To measure the time spent in other components, we use traces from a PCIe analyzer. Note that one can use this analysis infrastructure for any CPU or interconnect of interest.

The nodes in our setup are ThunderX2-based (TX2) servers and are connected with the TOP500-popular Mellanox InfiniBand [24] high-speed interconnect. Specifically, we use ConnectX-4, a recent Mellanox InfiniBand adapter, and attach it to the node through a PCIe slot. A Lecroy PCIe analyzer sits just before the NIC on node 1, as shown in Figure 2.4. The overhead of the PCIe analyzer is negligible as we did not observe any difference in performance with and without it. Larsen et al. observe the same [67]. The analyzer is a passive instrument that allows data to pass through fully unaltered [19].

For our software stack, we use the CH4 device of MPICH [87] with Unified Communication (UCX) [93] as the underlying communication framework. Specifically, we use UCX's accelerated *rc_x* transport which is UCX's implementation of the data-path operations, such as posting to the transmit queue and polling from the completion queue, for modern Mellanox InfiniBand adapters.

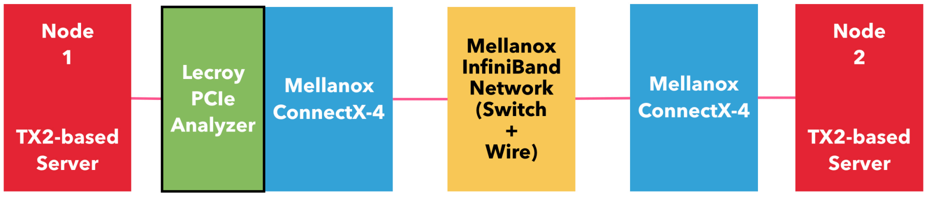


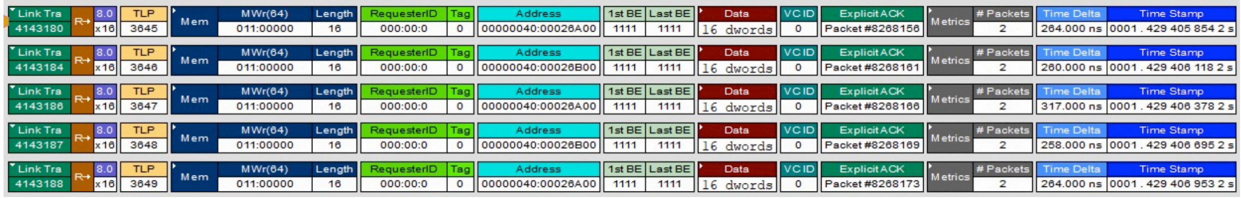
Figure 2.4: Two-node setup with PCIe analyzer on node 1.

2.3 Measurement Methodology

Software measurements. To measure time spent in the CPU, we wrap the relevant code with UCX's UCS profiling infrastructure [23], which, for `aarch64`, internally reads the `cntvct_el0` register timer preceded by an `isb`. The profiling infrastructure reads the timer at the start and end of the code of interest. Since the execution times of these regions of interest are in the order of nanoseconds, the overhead of the timer infrastructure needs to be accounted for. The mean overhead of UCS's profiling infrastructure on our evaluation setup is 49.69 nanoseconds (a standard deviation of 1.48 for 1000 samples); we report software measurements in the rest of this chapter after removing this overhead. While measuring time of a component, we do not simultaneously measure time in any other component to minimize any effects of artificial slowdowns caused by the timer infrastructure.

System measurements. We use the PCIe analyzer to measure the time spent in the hardware components such as the network and I/O components. We also use it to validate our software measurements wherever possible. The analyzer records a timestamp for each transaction that it traces (see Figure 2.5). Hence, to measure the time taken for an event, we take the delta of the timestamps of the PCIe activity that occur immediately after and immediately before the event of interest.

Each reported CPU or PCIe analyzer measurement is a mean of at least 100 samples.



Link Tra		8.0	TLP		MWr(64)	Length	RequesterID	Tag	Address	1st BE	Last BE	Data	VC ID	Explicit ACK		# Packets	Time Delta	Time Stamp
4143180	R→	x16	3645	Mem	011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268156	Metrics	2	264.000 ns	0001 . 429 405 854 2 s
4143184	R→	x16	3646	Mem	011:00000	16	000:00:0	0	00000040:00026B00	1111	1111	16 dwords	0	Packet #8268161	Metrics	2	260.000 ns	0001 . 429 406 118 2 s
4143186	R→	x16	3647	Mem	011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268166	Metrics	2	317.000 ns	0001 . 429 406 378 2 s
4143187	R→	x16	3648	Mem	011:00000	16	000:00:0	0	00000040:00026B00	1111	1111	16 dwords	0	Packet #8268169	Metrics	2	258.000 ns	0001 . 429 406 695 2 s
4143188	R→	x16	3649	Mem	011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268173	Metrics	2	264.000 ns	0001 . 429 406 953 2 s

Figure 2.5: PCIe trace of downstream PCIe transactions for UCX's RDMA-write injection-rate benchmark (`put_bw`).

2.4 Definitions

For our modeling and analysis in this chapter, we define two levels of communication components: low and high.

Low-level components include the low-level communication protocol (LLP), the I/O subsystem, and the network components. The LLP software drives the I/O and network hardware. In our study, UCX's low-level transport API, UC-Transports (UCT), is the LLP driver. UCT abstracts the capabilities of the various hardware architectures with minimal software overhead.

High-level components include high-level communication protocols (HLP). The most commonly used programming model for large-scale parallel systems today is MPI [97]. Modern MPI implementations, such as the CH4 device of MPICH, use abstract communication frameworks, such as UCX's UC-Protocols (UCP) layer, so that the MPI libraries do not need to maintain separate critical paths for all interconnects. UCP implements high-level communication protocols such as collectives, message fragmentation, etc. using the low transport-level capabilities exposed through UCT. MPI libraries use UCP to implement the specifications of the MPI standard.

We define variables to represent time in each component.

- *Post* – Total time spent in the initiation of an operation
- *Progress* – Total time spent during the progress of an operation
- *LLP_post* – LLP performing a PIO post of one 8-byte message
- *LLP_prog* – LLP dequeuing one entry of the completion queue during the progress of an operation

- *HLP_post* – Time spent in HLP during a *Post*
- *HLP_tx/rx_prog* – Time spent in the HLP during the progress of a send (TX) or receive (RX) operation
- *PCIe* – payload traversing PCIe between RC and NIC
- *Wire* – payload traversing the physical wire of the interconnect
- *Switch* – overhead added by a network switch
- *Network* – the total time in the interconnect (*Wire* + *Switch*)
- *RC-to-MEM(xB)* – RC writing an x-byte payload to memory

2.5 Overall Injection Overhead

Injection is the insertion of a message into the network. The message is injected when the payload reaches the NIC. We study the case when the user is transmitting messages continuously since this represents a system's injection limit. Then, the system's injection overhead, *Inj_overhead*, is the time difference between messages arriving at the NIC. This *Inj_overhead* explains why all the messages in a burst do not reach the NIC at time zero. We first model the injection overhead of PIO posts for a small message, then measure the overhead according to the model, and finally validate it for an Arm-based server.

2.5.1 Modeling

Since the depth of the transmit queue (TxQ) is finite, the user cannot post indefinitely. Polling the completion queue (CQ) serves as the dequeue semantic for the TxQ. Hence, the user must poll in between their posts to inject messages into the NIC. Say, the user polls after every p posts. If $p = 1$, the depth of the TxQ is not utilized and the post translates to a synchronous post, that is, the user will be able to post the next message only after the previous message has reached the target node (since the completion is generated only when the host NIC receives an ACK from the target NIC (see Section 2.1.3)).

To remove the overhead of waiting for a previous message to complete, the user must choose a value of p such that the completion for an earlier message is available during a poll. Such a value of p depends on the value of *Post* and the time taken to generate a completion, $gen\_completion$. From Section 2.1.3, we can deduce that

$$gen\_completion = 2 \times (\textit{PCIe} + \textit{Network}) + \textit{RC-to-MEM(64B)} \tag{2.1}$$

since the PCIe wire and the interconnect's network fabric are traversed twice: first while transmitting the message to the target NIC, and second while receiving the ACK from the target NIC and writing the corresponding completion. A completion in InfiniBand is 64 bytes and hence, the RC conducts a 64-byte write to memory on behalf of the host NIC. Then, to remove the overhead of waiting for a previous message, the lower bound on p is

$$p \geq gen\_completion/\textit{Post} \tag{2.2}$$

In our modeling of the injection overhead, we assume that the user meets this lower bound on p.

Typically, the API of the network driver allows the user to poll a batch of completions, reducing the overhead of expensive memory barriers and function calls [66]. Say the user polls b number of completions in each batch. This means that the user can post only b posts in the next round of posts since only b entries have been dequeued from the TxQ. Note that b meets the lower bound mentioned above. Additionally, the user could perform some miscellaneous operations during the window of b posts or b polls. Let tot_misc demarcate the cumulative time spent in these other operations. Then, the

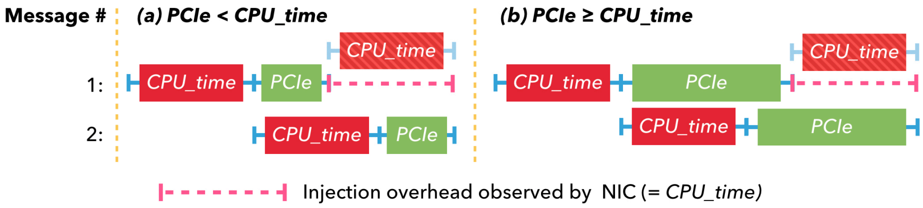


Figure 2.6: Injection overhead observed by the NIC.

overhead of the CPU to post a message is

$$\begin{aligned} \mathit{CPU\_time} &= \frac{b \times \mathit{Post} + b \times \mathit{Progress} + \mathit{tot\_misc}}{b} \\ &= \mathit{Post} + \mathit{Progress} + \mathit{Misc} \end{aligned} \tag{2.3}$$

where $\mathit{Misc} = \mathit{tot\_misc}/b$ is the miscellaneous overhead amortized for each message.

Hence, on average, messages arrive at the RC every *CPU_time*. Since PCIe supports multiple outstanding requests (see Section 2.1.3), the RC initiates MWr PCIe transactions targeting the NIC as soon as it receives messages from the CPU. Considering that the RC is implemented with hardware logic, the time it takes to generate a transaction would be in the order of a few cycles. Hence, we ignore its contribution to the injection overhead. Note that the RC can generate transactions only if it has enough credits. Otherwise, it needs to wait for an UpdateFC DLLP from the NIC which would incur the overhead of the PCIe wire between the NIC and the RC (*PCIe*). Experientially, we observe that a single core does not exhaust the credits for MWr transactions. Hence, we do not model for the overheads imposed with exhausted credits in this chapter.

Once the message leaves the RC, it incurs *PCIe* before arriving at the NIC. Hence, the injection overhead of a *single* message is

$$\mathit{Msg\_inj\_overhead} = \mathit{CPU\_time} + \mathit{PCIe} \tag{2.4}$$

While *Msg_inj_overhead* describes the time taken by each message to reach the NIC, it is not the same as the injection overhead observed by the NIC, *Inj_overhead*, as we shall see next. When the system is issuing messages continuously, the *CPU_time* of the next message overlaps with the *PCIe* of the previous one (see Figure 2.6). Hence, the time difference between the initiation of messages is *CPU_time*. This holds true for any relation of *PCIe* with *CPU_time* (assuming that *PCIe* is not long enough to exhaust the RC's credits). When $PCIe > CPU\_time$, *PCIe* of the next message can also overlap with *PCIe* of the previous one. Hence, from the perspective of the NIC, the time difference between the arrival of messages is the same as that between the initiation of messages, that is,

$$\begin{aligned} Inj\_overhead &= CPU\_time \\ &= Post + Progress + Misc \end{aligned} \tag{2.5}$$

2.5.2 Breakdown

Given the modeled injection overhead of a system (*Inj_overhead* in Equation 2.5), we measure its breakdown for the Arm-based server considering both low and high level components and their interactions. *Post* is simply the sum of the times spent in the high-level communication protocol (HLP) and the low-level communication protocol (LLP) to initiate a send operation, that is, *Post* = *LLP_post* + *HLP_post*. *Progress* is the total overhead imposed by both the HLP and LLP for the progress of the send operation. Unlike *Post*, however, *Progress* is not a simple sum of *LLP_prog* and *HLP_tx_prog* because of certain performance-oriented optimizations which result in complex mechanisms.

We measure the times for the low and high-level components separately and then combine them together to provide the complete breakdown. Finally, we provide key insights from the breakdown of the overall injection overhead.

Low-level Components

Using the low-level UCT driver, we run the UCX `perftest`'s injection-rate microbenchmark, namely the `put_bw` test, with a single core. Each message is 8 bytes, the size of a `double`. By involving just the low-level components, Equation 2.5 becomes

$$\begin{aligned} Inj\_overhead &= CPU\_time \\ &= LLP\_post + LLP\_prog + Misc \end{aligned} \tag{2.6}$$

We measure *LLP_post*, *LLP_prog*, and *Misc* by wrapping the UCS profiling infrastructure around the calls to `uct_ep_put_short`, `uct_worker_progress`, and the relevant regions of the microbenchmark, respectively. Table 2.1 reports our measurements and Figure 2.7 shows a percentage breakdown of *Inj_overhead* when only low-level components are involved. *LLP_post* constitutes more than half of the the injection overhead. We describe what occurs in the different components next.

LLP_post. The user-space network driver prepares the appropriate resources and writes directly to the NIC's registers without any involvement of the kernel. The following details the steps involved in an *LLP_post*.

1. Prepare message descriptor (MD) – this involves the time taken to write the control segment of the descriptor. It also involves a `memcpy` of the small payload when inlining is used.

2. A store memory barrier – this ensures that the MD is completely written before the CPU signals the NIC. This barrier is relevant only for a weak memory model (`dmb st` on `aarch64`).

3. *DoorBell* counter increment – the NIC reads a *DoorBell* counter to perform speculative reads. The CPU updates this counter before writing to the NIC.

4. A store memory barrier – this ensures that the NIC sees the update to the *DoorBell* counter before any subsequent write to its device memory.

5. PIO copy – this is the CPU's write to the memory-mapped device memory instructing the NIC to transmit the message. Device memory is typically an uncached, buffered memory region that supports out-of-order writes. For the TX2-based server in our setup, we use Device-GRE memory for the memory-mapped location. Though there would be a store memory barrier (`dsb st`) after the PIO copy to flush the data to the NIC, we observed experientially that this flush is not necessary for the microarchitecture of the TX2-based server. The PIO copy of an 8-byte message is one 64-byte chunk in Mellanox InfiniBand (see Section 2.1.3).

To measure the breakdown of the time spent in the different sections of the LLP, we wrap the relevant regions of code in the implementation of `uct_ep_put_short`. While the categories listed above are critical components of an *LLP_post*, they do not account for other miscellaneous time such as the function call overhead, branches to decide code path, etc. We compute this time by taking the difference of *LLP_post* and the sum of the times spent in the categories. Table 2.1 reports our measurements and Figure 2.8 shows the breakdown of an *LLP_post* for the TX2-based server. PIO constitutes the majority of *LLP_post*. We discuss it later in Section 2.7.1 in the context of simulated optimizations.

LLP_prog. During the progress of an operation the LLP reads the designated memory location (where the NIC DMA-writes its completions). This progress operation constitutes a load memory barrier for `aarch64`'s weak memory model to ensure that the read for a completion queue entry occurs before subsequent updates to data structures.

Misc. Since miscellaneous time between consecutive posts is specific to the benchmark, we study the behavior of the microbenchmark and measure the miscellaneous time accordingly. This time is reported in Table 2.1.

Table 2.1: Measured times of low-level software components.

Component	Time (ns)
Message descriptor (MD) setup	27.78
Barrier for message descriptor (MD)	17.33
Barrier for *DoorBell* counter (DBC)	21.07
PIO copy (64 bytes)	94.25
Other in *LLP_post*	14.99
LLP_post (total of above)	175.42
LLP_prog	61.63
Misc in $Inj\_overhead$	58.68

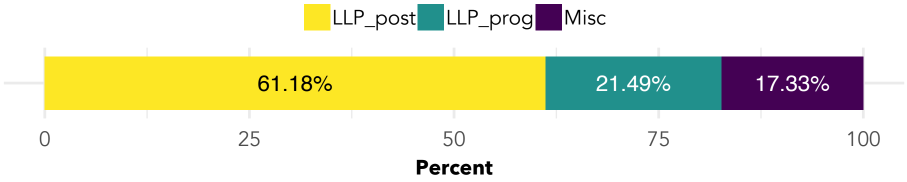


Figure 2.7: Breakdown of injection overhead with the LLP.

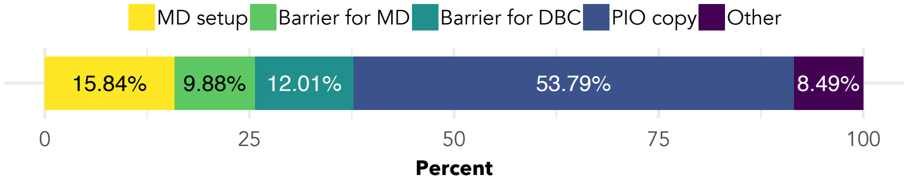


Figure 2.8: Breakdown of time in an *LLP_post* (MD: message descriptor; DBC: *DoorBell* counter).

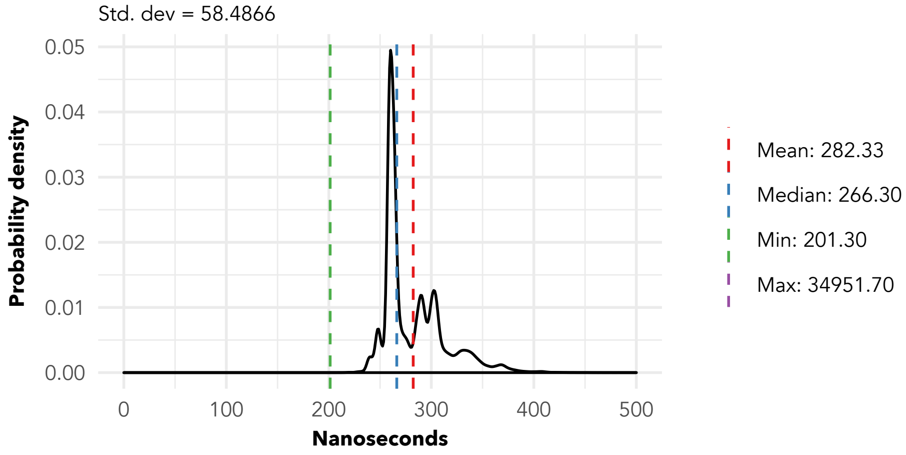


Figure 2.9: Distribution of the observed injection overhead (maximum value is an outlier).

Validating modeled injection overhead. Since the PCIe analyzer sits just before the NIC, the timestamps of the 64-byte PIO posts correspond to the times at which the messages reach the NIC. Hence, calculating the delta of the timestamp of consecutive transactions results in the observed *Inj_overhead*. Figure 2.9 shows the distribution of this overhead. The modeled injection overhead (**295.73** nanoseconds) is within **5%** of the mean observed injection overhead (**282.33** nanoseconds).

High-level Components

We present a breakdown of time spent in the high-level communication protocol (HLP) for a communication-initiation operation such as an `MPI_Isend`, and a communication-progress operation such as a successful (*i.e.* no busy-waiting) `MPI_Wait` corresponding to an `MPI_Irecv`.

Communication initiation. In an `MPI_Isend`, the MPI library first decides how to best execute the operation by checking if the data is contiguous, computing which communication interface to use, etc. Ultimately it calls into the UCP layer (`ucp_tag_send_nb`) which eventually executes the LLP in the UCT layer (`uct_ep_am_short`). To measure the time spent in MPI and UCP for an `MPI_Isend`, we first measure the total time of `MPI_Isend`, the total time of `ucp_tag_send_nb` inside MPICH, and the total time of `uct_ep_am_short` inside UCP by wrapping them with the UCS profiling infrastructure. We can then measure the time spent in MPI and UCP by taking the differences of times between the upper and lower layers. For example, subtracting the total time of `ucp_tag_send_nb` from that of `MPI_Isend` gives us the time spent in MPI.

Communication progress. In an `MPI_Wait`, the MPI library executes its progress engine which ultimately calls into the UCP layer (`ucp_worker_progress`). UCP then ensures progress on all outstanding operations that have been posted by progressing the low-level

Table 2.2: Measured times of various components.

Component	Time (ns)
MPI_Isend in MPICH	24.37
MPI_Isend in UCP	2.19
Callback for a completed MPI_Irecv in MPICH	47.99
Successful MPI_Wait for MPI_Irecv in MPICH	293.29
Callback for a completed MPI_Irecv in UCP	139.78
Successful MPI_Wait for MPI_Irecv in UCP	150.51

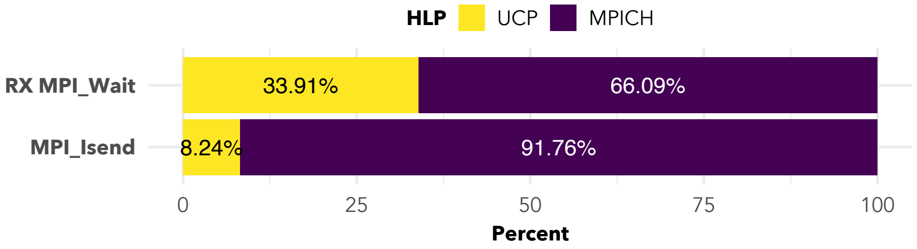


Figure 2.10: Breakdown of time in HLP.

UCT layer (uct_worker_progress). When an operation completes, UCT executes a registered callback into the upper UCP layer to update data structures that indicate the completion of the operation. Similarly, the UCP callback also executes a registered callback into the upper MPI layer to indicate that the operation has completed. Note that these callbacks are executed before returning from uct_worker_progress. To measure the time spent in MPICH and UCP for an MPI_Wait, we measure the times spent in the registered MPICH and UCP callbacks in addition to measuring the total times of MPI_Wait, ucp_worker_progress, and uct_worker_progress. Since the UCP callback entails the MPI callback, we measure the time spent in the UCP callback only by taking the difference between the total times spent in the callbacks. We then measure the time spent in MPI and UCP by taking the differences of times between the upper and lower layers and adding in the time for the upper layer's registered callback. For example, subtracting the total time of ucp_worker_progress from that of MPI_Wait and adding in the time of the MPI callback yields the time spent in MPI [108].

Table 2.2 reports the time spent in MPI and UCP on top of the LLP's HW/SW interface for an `MPI_Isend` (26.56 nanoseconds in total), and a successful `MPI_Wait` for an `MPI_Irecv` (443.8 nanoseconds in total). Figure 2.10 shows their percentage breakdown.

The Complete Picture

As mentioned earlier, *Post* in Equation 2.5 is the the sum of *LLP_post* and *HLP_post* (201.98 nanoseconds). *Progress*, on the other hand, is not a straightforward sum of the low and high level components because of their complex interactions. We describe the optimizations responsible for the complex behavior.

UCP reduces the overhead of progressing send operations by using unsignaled completions, which means the NIC DMA-writes a completion only every c operations to indicate the completion of all c operations ($c = 64$ in UCX). Hence, the overhead of progress is amortized over c operations. Depending on the value of c and the depth of the underlying transmit queue (TxQ), certain posts of the LLP can fail due to a full TxQ. UCT defers the successful execution of such ″busy posts″ during the progress of operations implying that the progress of some operations includes the overhead of initiation in LLP (*LLP_post*). We carefully account for such complex interactions between the components and measure *Progress* and *Misc* to be 59.82 nanoseconds and 3.17 nanoseconds, respectively. Less than a nanosecond of *Progress* occurs in the LLP (due to the amortization); the rest occurs in the HLP (*HLP_tx_prog*).

Validating modeled injection overhead. In Section 2.5.2 we validated that a microbenchmark can accurately measure the true injection overhead of a system. Hence, we use OSU Micro-Benchmark's [22] message rate test[1] to measure the observed injection overhead. By taking the inverse of the message rate, we measure the mean injection overhead to

[1] We remove the send-receive sync after every window of posts for a clear analysis.

be **263.91** nanoseconds. The injection overhead computed with Equation 2.5 is **264.97** nanoseconds which is within **1%** of the observed overhead. Figure 2.11 shows the breakdown of the overall injection overhead.

Key Insights

Insight 1. An application cannot indefinitely initiate messages due to the limited depth of the transmit/receive queue; the progress of an operation serves as a ”semantic bottleneck.” However, once optimizations like unsignaled completions for send operations minimize the performance overheads imposed by this bottleneck, Figure 2.11 shows that *Post* dominates (more than 70% of total) the overall injection overhead. Within *Post*, the low-level communication protocol (LLP) dominates as seen in ”Initiation” of Figure 2.12.

Insight 2. Figure 2.12 shows that the HLP dominates the progress of both send and re-

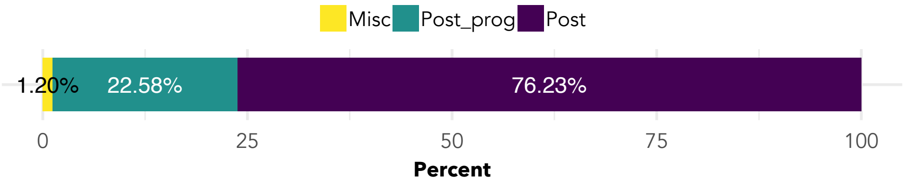


Figure 2.11: Breakdown of the overall injection overhead.

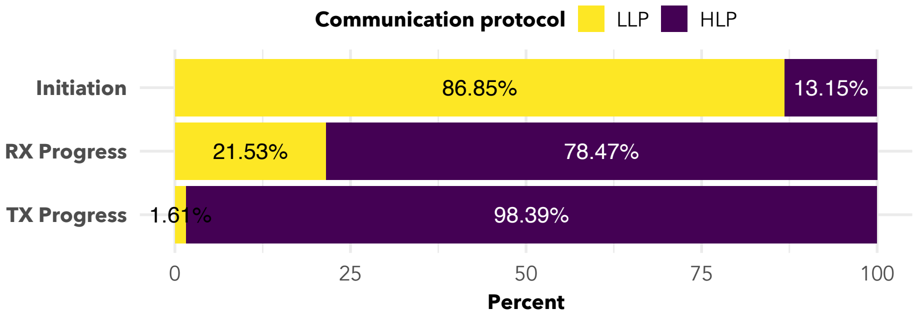


Figure 2.12: Breakdown of time in HLP and LLP during the initiation and progress of communication.

ceive operations. Furthermore, the progress of a receive operation is $4.78\times$ higher than that of a send operation.

2.6 End-to-End Latency

Latency is the total time incurred by a message starting from the time the host node initiates the transfer to the time of writing the payload in the destination buffer on the target node.

2.6.1 Modeling

We study the latency of a short message transmitted using send-receive semantics. The initiation of the transmission begins with an *Post*, after which the message traverses the PCIe fabric and reaches the NIC. The NIC then transmits the message over the network fabric to reach the target node. On the target node, the NIC performs a MWr PCIe transaction, which traverses the PCIe wire and instructs the RC to write the payload into the target node's memory. Meanwhile the CPU on the target node has been polling for its posted receive to complete. The user can use its receive buffer only after a successful poll. Thus, for a payload of size, x, the time for latency is derived as follows,

$$Latency = Post + 2(PCIe) + Network + \textit{RC-to-MEM(xB)} + Progress \tag{2.7}$$

2.6.2 Breakdown

Considering the low and high level components involved in the latency of a system, Equation 2.7 becomes

Table 2.3: Measured times of system hardware components.

Component	Time (ns)
PCIe for a 64-byte payload	137.49
Wire	274.81
Switch	108
Network (total of above)	382.81
RC-to-MEM(8B)	240.96

$$
\begin{aligned}
\mathit{Latency} = {} & \mathit{HLP\_post} + \mathit{LLP\_post} + 2(\mathit{PCIe}) + \mathit{Network} \\
& + \textit{RC-to-MEM(xB)} + \mathit{LLP\_prog} + \mathit{HLP\_rx\_prog}
\end{aligned}
\tag{2.8}
$$

We measure the time for each component according to Equation 2.8 to achieve the breakdown of the latency of the two-node TX2-based system.

Low-level Components

The values of *LLP_post* and *LLP_prog* in Equation 2.8 are the same as the ones measured in Section 2.5.2. Table 2.3 contains the times for the rest of the system components. We describe how to measure them next.

PCIe. To measure *PCIe*, we first measure the round-trip latency of the PCIe wire between the NIC and the RC. Since the PCIe analyzer sits just before the NIC, any transaction initiated by the NIC and the corresponding ACK DLLP from the RC will give us the start and end time of the required round-trip. For this purpose, we use the MWr transactions initiated by the NIC during the DMA-write of completions. The timestamp in the MWr transaction is the start time of the round trip and that in the corresponding ACK DLLP is the end time. Dividing this round-trip value by two is *PCIe* (the size of this MWr transaction is the same as that of the PIO copy: 64 bytes).

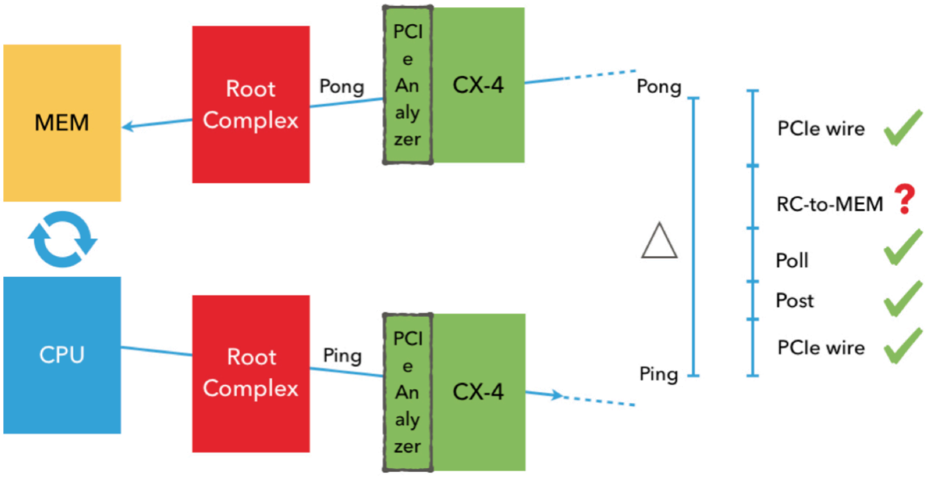


Figure 2.13: Measuring *RC-to-MEM(xB)* using the time delta between an inbound pong and outbound ping on node 1.

Network. One way to measure *Network* would be to first measure the time difference between when a PIO post reaches the NIC and when the NIC receives an ACK from the target node for that PIO post. Then, dividing that difference by two would correspond to *Network* since the difference entails a round-trip *Network* latency. The timestamps on the PCIe trace of the ping-pong style `am_lat` benchmark allow us to employ this method. A downstream 64-byte PCIe transaction corresponds to a ping and the next upstream 64-byte PCIe transaction corresponds to the ping's completion which is generated upon reception of the ACK. Doing so, we measured the value of *Wire* to be 274.81 nanoseconds for a direct NIC-to-NIC connection. If the NICs are connected via a switch, the overhead of *Switch* is 108 nanoseconds. We measured this overhead by taking the difference between two latency measurements: one with a switch involved and one without.

RC-to-MEM(8B). To measure *RC-to-MEM(8B)*, we utilize the timestamps on the PCIe trace data of the `am_lat` ping-pong benchmark. As shown in Figure 2.13, the time difference between an incoming pong and outgoing ping entails an *RC-to-MEM(8B)*, two *PCIe*s (one for the inbound pong and the other for the outbound ping), a *LLP_prog* (successful poll), and a *LLP_post* (the ping). Once we measure the pong-ping difference from the PCIe trace, we can compute the value of *RC-to-MEM(8B)* since we have measured the values of the other components.

Validating modeled latency. Plugging in the measured values into Equation 2.8 we obtain *Latency* = **1135.8** nanoseconds. The observed latency from UCX's `am_lat` test is **1190.25** nanoseconds, which is within 5% of the modeled latency.

High-level Components

HLP_post in Equation 2.8 is the same as the one in Section 2.5.2. *HLP_rx_prog* in Equation 2.8 is the sum of the times spent in the registered callbacks of the MPI library and UCP along with the remaining time spent in the MPI library after `ucp_worker_progress` returns. Note that the latter is not the equivalent of the total time spent in MPI for a successful `MPI_Wait` minus the time spent in the MPI library's callback. `MPI_Wait` is a blocking call and incurs a portion of its 293.99 nanoseconds before even progressing UCP. MPICH internally loops on `ucp_worker_progress` until the operation is complete. Hence, we specifically measure the time spent in MPICH after a successful `ucp_worker_progress` and observe this time to be 36.89 nanoseconds. Adding this time to the times spent in the callbacks of MPICH and UCP reported in Table 2.2, the value of *HLP_rx_prog* is 224.66 nanoseconds.

The Complete Picture

Summing up the values of *LLP_prog*, *HLP_post*, and *HLP_rx_prog* to the modeled latency in Section 2.6.2, the end-to-end latency is **1387.02** nanoseconds. This is within **4%** of the observed latency of **1336** nanoseconds measured by OSU Micro-Benchmark's point-to-point latency test. Figure 2.14 shows a detailed breakdown of this latency.

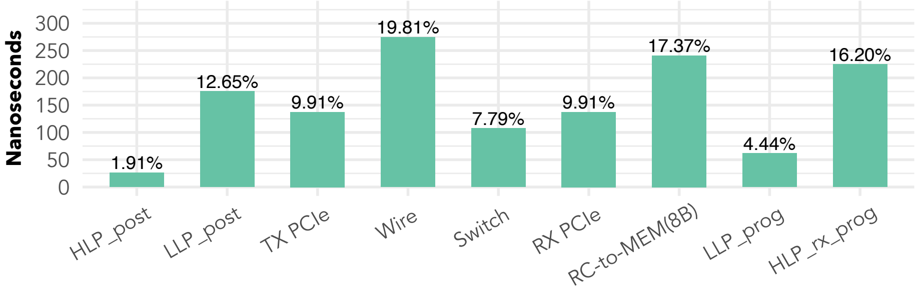


Figure 2.14: Breakdown of the end-to-end latency.

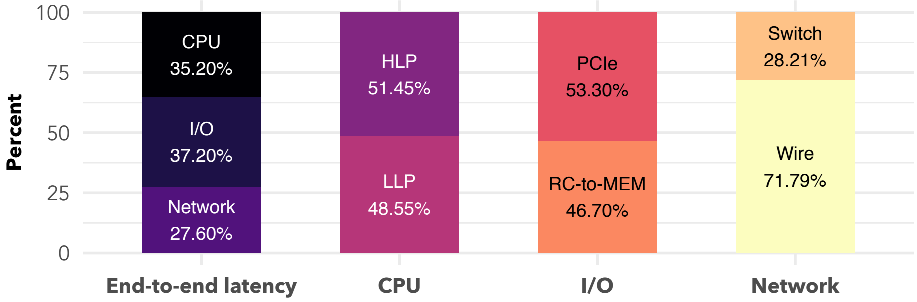


Figure 2.15: High-level breakdown of the end-to-end latency.

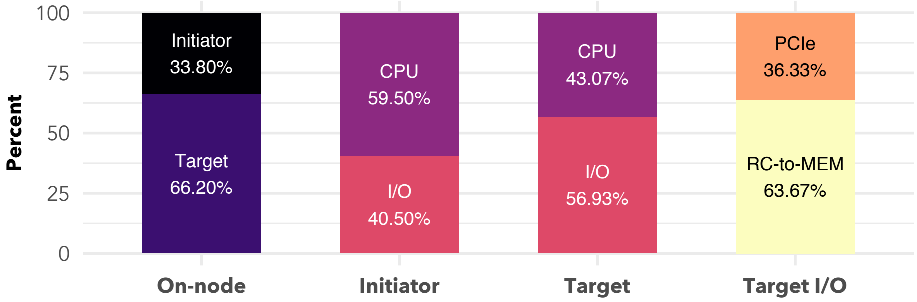


Figure 2.16: Breakdown of time spent on node.

Key Insights

Insight 3. Figure 2.15 presents the overall percentage breakdown of the end-to-end latency of a small message in the three categories: CPU, I/O, and network. The constituents of the software and I/O categories contribute almost equally (within 4% of each other) to their respective total times. In the case of *Network*, the latency of *Wire* dominates the overall off-node time. Note that none of the three categories dominates the overall latency. However, we observe that the network fabric constitutes less than a third of the overall latency while CPU and I/O components together contribute towards 72.4% of the latency. Hence, most of the overhead in the transmission of a small message is incurred on the node.

Insight 4. Figure 2.16 shows a high-level breakdown of the time spent on the node during the transmission of the message. The majority of this time occurs on the target node. Out of the time on the target node, the majority occurs during I/O, the majority of which is comprised by the RC writing the payload to memory. On the contrary, software comprises the majority of the time spent on the initiator node. This is due to the use of Programmed I/O (see Section 2.1.3) for short messages. Consequently, I/O on the initiator node comprises only of a PCIe transaction unlike that on the target node.

2.7 Simulated Optimizations

In this section, we use the insights gained from the breakdown of the overall injection overhead and the end-to-end latency of a system to study the effects of optimizing the CPU, I/O, and network fabric components on the injection and latency of small message transfers. In the figures that follow, we aim to answer the following question: if we optimize component X by Y%, what is the corresponding reduction in injection overhead and latency? The horizontal axes of Figures 2.17 and 2.18 represent the degree of optimization for the component of interest. It consists of five evenly spaced reductions in overhead, starting from 10% ($1.1\times$ faster) to 90% ($10\times$ faster). The vertical axis represents the speedup in the overall injection or end-to-end latency as a result of reducing the component's overhead. Such a way to simulate optimizations has provided key insights in a wide variety of scenarios [83, 84, 82]. Note that the components of our models are not concurrent, that is, their executions do not overlap. Hence, evaluating the impacts of reductions in overheads on benchmarks such an MPI stencil kernel through a distributed system simulator (such as SimGrid [39]) results in exactly the same linear speedups that we generate through a manual what-if analysis in Figures 2.17 and 2.18.

We organize our discussion into a set of relevant optimizations that target the different components. For each optimization we discuss their likelihood and evaluate their impact. We consider speedups more than 5% to be substantial.

2.7.1 On-node optimizations

We learned in Section 2.6.2 that most of the time in the transmission of a small message is spent on the node (insight 3). CPU and I/O components make for the on-node time. Below we discuss three relevant optimizations.

NIC integrated into a System-on-Chip

The idea of this optimization is that the NIC sits on the same die as that of the processor. The deployment of such a solution would be in the form of a System-on-Chip (SoC) so that instead of interfacing with the CPU through the PCIe subsystem, the NIC would connect to the network-on-chip (NoC). Such a tight integration of the NIC and the CPU would eliminate a majority of the I/O subsystem's overhead, which accounts for the majority of the time in the latency of a small message. While integrated NICs are not commonplace in today's HPC systems, they are more than likely to become ubiquitous in the future given the potential of their impact. There have been multiple works [37, 68] that argue for and evaluate the performance of SoC-integrated NICs showing their benefits in terms of better performance and higher CPU availability for all message sizes. More recently, Arm-based supercomputers are on the rise [62] since they allow HPC vendors to integrate their custom solutions (such as an integrated NIC) with Arm IP on SoCs. The Tofu interconnect D [25] on Fujitsu's Fugaku machine is a prominent example of this optimization. With Tofu's NIC integrated into a Fugaku node, the RDMA-write latency has been improved by nearly 400 nanoseconds.

Impact. "Integrated NIC" in Figure 2.18b shows the impact of a solution that simply brings the NIC closer to the TX2-based SoC. While one can expect such a solution to eliminate most of the I/O overhead, we can observe over a 15% improvement in overall latency even with a modest 50% reduction in I/O time. In fact, a tightly integrated NIC allows for opportunities to reduce the involvement of the CPU in the LLP's HW/SW interface and thereby increase its availability for computational tasks. Recall that the reason for the use of PIO for small messages is expensive PCIe round-trip latencies with the communication-offloading approach (see Section 2.1). Since an integrated NIC would sit close to memory, round-trip latencies performed by the hardware logic of the NIC would most likely be faster than involving the CPU in PIO.

Improving the initiation of a message in the low-level communication protocol

This optimization deals with how writes to device memory occur in the microarchitecture of a processor. Ideally, writes to `aarch64`'s *Device memory* [71] should be as fast as writes to its *Normal memory* [71]. Such an optimization is likely since the current difference between 64-byte writes to Normal and Device memory is more than 90%, hinting that there exists room for optimization in the TX2-based server. Optimization efforts would reduce the time spent in the PIO copy, which accounts for more than 50% of the time in *LLP_post* (see Figure 2.8). Researchers at Arm have used the results of this study to improve the latency of writes to Device Memory in the microarchitecture of the server's next iteration: ThunderX3.

Impact. "PIO" in Figure 2.17 and Figure 2.18a shows the impact of improving the 64-byte PIO copy on the overall injection and end-to-end latency, respectively. A regular 64-byte `memcpy` on the TX2-based server takes less than a nanosecond as expected. If we modestly project the overhead of PIO to reduce to 15 nanoseconds (84% reduction), overall injection can improve by more than 25% and end-to-end latency can improve by more than 5%.

Reducing software overheads

This optimization deals with software engineering targeted to reduce overheads in the HLP. However, unlike the previous optimizations, it is unlikely that this optimization would reduce overheads by more than 50%. For example, the current implementation of MPICH is highly optimized [87], reducing the number of instructions by 76% from its previous implementation for an `MPI_Isend`. We conjecture that software optimizations would reduce overheads by less than 20%.

Impact. Figure 2.17 and Figure 2.18a show the what-if analysis for the different constituents in the HLP and LLP. The "HLP" and "LLP" lines in the figures reflect the upper bound on speedups that would result from optimizing the components that constitute the HLP and LLP, respectively. For both injection and latency, optimizing the progress of operations in the HLP (*HLP_tx_prog* and *HLP_rx_prog*) can achieve speedups close to HLP's

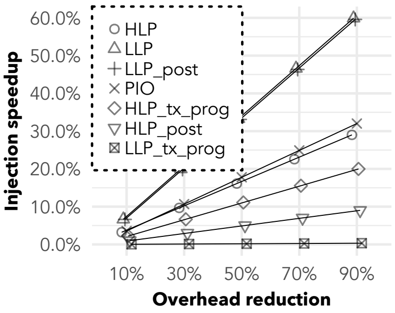
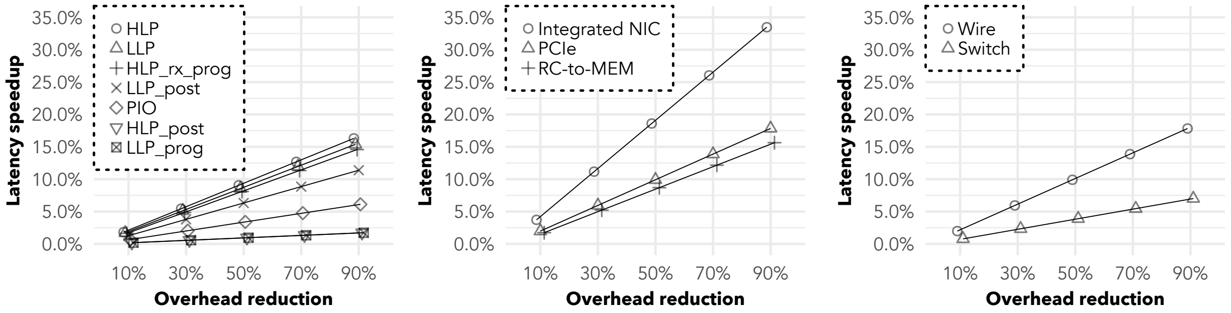


Figure 2.17: Simulated speedups in overall injection by reducing CPU overheads.



(a) CPU (b) I/O (c) Network

Figure 2.18: Simulated speedups in end-to-end latency by reducing overheads of CPU, I/O, and network components.

upper bound. Similarly, optimizing *LLP_post* can achieve speedups close to LLP's upper bound. If we consider software overheads would be reduced at most by 20%, the upper bounds reflect a less than 5% speedup in the end-to-end latency. On the other hand, a 20% reduction in overhead in the LLP can speedup injection by up to 13.33% while that in the HLP can do so by up to 6.44%.

2.7.2 Off-node optimizations

Figure 2.15 shows that 27.6% of the end-to-end latency is spent on the interconnect's *Wire* and in the *Switch*. Our foresight is that the reduction in off-node overheads is less than likely and that the resulting speedups with off-node optimizations alone would not be substantial. We explain our foresight below.

The reduction in *Wire*'s overhead is less than likely due to engineering complexities at the physical layer. In fact, it is possible that the latency will increase in future interconnects in order to accommodate for higher throughput. The conversion between the parallel PCIe signals and the serial signals on the interconnect's fiber transmission link occurs through SerDes (serializer/deserializer) integrated circuits. For throughputs higher than 100 Gb/s, the SerDes unit needs to be able to deliver higher throughput. While higher degrees of pulse amplitude modulation (PAM) deliver higher signal rates, they require more complex forward error correction (FEC), which increases the latency of the transmission in some cases by 300 nanoseconds [96, 48, 36].

The current latency of a high-performance interconnect's switch is already an order of magnitude lower than that of an Ethernet's switch [90]. New technologies like GenZ forecast their switch latencies to be 30-50 nanoseconds [40]. However, such low latencies are yet to be demonstrated. Only an optimistic reduction to 30 nanoseconds (72% overhead reduction) would correspond to a substantial speedup (5.45%) in end-to-end latency according to Figure 2.18c.

2.8 Concluding Remarks

Our injection-overhead and latency models accurately describe high performance communication. The communication performance breakdown of a system corresponding to our model explains where and how much time is spent during communication. Like it did for the engineers of the Arm-based server, a breakdown guides the optimization efforts of system architects. Given that the communication performance of non-traditional server-class CPUs is slower than that of traditional ones (see Figure 2.1), we encourage the researchers and engineers to use our detailed methodology to produce a communication performance breakdown for systems of their own interest.

Future work. The models in this chapter are restricted to the case of a single core driving communication. In a multithreaded environment, factors such as limited PCIe credits and barrier synchronization have the potential to be more expensive. In the future, we would like to extend our study to model the communication performance in a multithreaded environment. Furthermore, from our discussions with scientists at various industry research institutions, we learned that PCIe analyzers are typically not readily available because of their high costs (in the order of million US dollars). Certain Intel Xeon processors provide performance counters to monitor PCIe traffic. Capturing PCIe behavior without a PCIe analyzer but with PCIe-related counters on a processor for the purposes of measuring time spent in system components remains an open question.

Chapter 3

Multithreaded Communication Capabilities on Modern Networks

Modern high performance interconnects, such as Mellanox InfiniBand, Intel Omni-Path, and Cray Slingshot, feature multiple network hardware contexts on each network interface card (NIC) attached to a node. They do so to accommodate parallel communication from the many cores on a node. Each hardware communication context serves as an interface into the network. So, with multiple network hardware contexts, each NIC enables parallel communication into the network from a single node.

The multiple network hardware contexts are transparently utilized when multiple processes exist since each process creates its own communication channel, but in a multithreaded environment each process needs to actively create multiple communication channels to map to the multiple network hardware resources on the NIC. As discussed in Chapter 1, MPI libraries today do not actively create multiple network communication channels. While this design drastically hurts the performance of multithreaded MPI communication, it is the most efficient in terms of resource usage. Middlewares, such as an

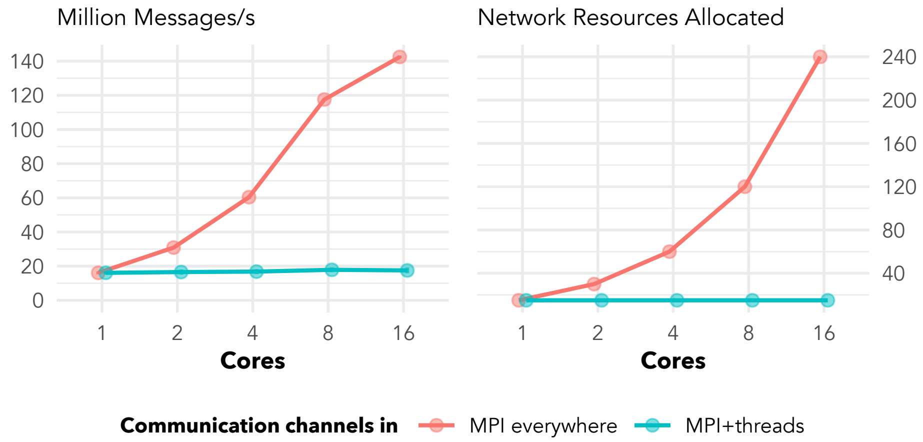


Figure 3.1: Throughput (higher is better) and network resource usage (lower is better) in state-of-the-art communication channels on Mellanox's ConnectX-4 adapter.

MPI library, need to take into account both the performance needs of an application and the availability of resources while creating multiple communication channels.

In this chapter, we study the capabilities of modern network hardware for the purpose of developing middleware that can manage multithreaded communication efficiently. We first study what exactly a network communication channel is, and what resources are impacted when a new communication channel is created. Given that a multithreaded environment, unlike a multiprocess environment, enables the sharing of communication resources between cores, we also study the performance impact of sharing different sets of communication resources between threads. Our study shows that the designs in state-of-the-art MPI everywhere and MPI+threads implementations represent extreme ends of the performance versus resource-efficiency tradeoff space and that multiple levels of sharing exist in between each with their own performance and resource usage characteristics.

We study RDMA-enabled NICs since they are used widely in the interconnects of HPC systems. RDMA NICs provide various performance advantages over the traditional Ethernet NICs: user-level access to the NIC to eliminate the overheads of expensive context switches to the kernel, and bypass of the remote CPU to access a remote node's memory. In particular, we study the capabilities of the Nvidia-Mellanox ConnectX series of adapters which are used both for InfiniBand and new RoCE (RDMA over Converged Ethernet) interconnects such as HPE Cray Slingshot [41].

3.1 Background

3.1.1 RDMA's Software Stack

The hardware-agnostic API to communicate over RDMA NICs is Verbs. As part of the InfiniBand standard, Verbs is a network-level API that is used by MPI libraries and, more recently, portable communication frameworks such as UCX and OFI. The network driver that implements Verbs for recent Nvidia-Mellanox ConnectX adapters (ConnectX-4 and later) is mlx5.

3.1.2 Verbs Communication Objects

The software bidirectional communication portal in Verbs is the queue pair (QP): a pair of send and receive FIFO queues, to which work queue entries (WQEs), Verbs' message descriptors, are posted. Each QP is associated with a completion queue (CQ) that contains completion queue entries (CQEs) corresponding to the completion of signaled WQEs. To create a QP, we need at least one memory buffer (BUF), device context (CTX), protection domain (PD), and CQ. A memory region (MR) is required if the NIC needs direct access to memory (as is the case in RDMA operations such as reads and writes). Additionally, we can assign QPs to thread domains (TDs) to provide single-threaded access hints to the QPs in a TD. Chapter 10 of the InfiniBand specification details the Verbs objects [5].

The CTX is the container of all Verbs objects and is also a slice of the network hardware, containing a subset of the NIC's hardware resources. In mlx5 devices, the hardware resources are part of the user access region (UAR) of the NIC's address space. Each UAR page consists of two micro UARs (uUARs)[1]. By default, a CTX contains eight UARs (UAR pages) and, hence, 16 uUARs. The user's QPs are mapped to one of the *statically allocated* uUARs unless a QP is part of a TD in which case the QP is mapped to a uUAR in a UAR that was *dynamically allocated* during TD creation. Appendix B exemplifies the QP-to-uUAR assignment policy for the ConnectX Nvidia-Mellanox adapters.

3.1.3 RDMA Operational Features

Verbs features two types of operations: one-sided *verbs* and two-sided *verbs*. The former is for RDMA operations such as reads, writes, and atomics which bypass the target CPU. The latter is for send-receive style messaging where the target CPU informs the NIC, using a receive WQE, where to write an incoming payload.

When the application executes a verb (e.g., `ibv_post_send`), what follows is a series of coordinated operations between the CPU and the NIC to fetch the WQE (DMA read), read its payload (DMA read), and signal its completion (DMA write). Section 2.1.3 of Chapter 2 describes these mechanisms. As we learned there, the overhead of multiple PCIe roundtrip latencies is hurtful especially for small-message communication. RDMA-NICs provide certain operational features to reduce the overheads of these mechanisms. While we have described some of them in Section 2.1.3, we reiterate them here in the context of Verbs and Nvidia-Mellanox hardware. Consider the depth of the QP to be n.

Postlist. Instead of posting only one WQE per `ibv_post_send`, Verbs allows the application to post a linked list of WQEs with just one call to `ibv_post_send`. It can reduce

[1] Appendix A provides further details about the anatomy of a UAR

the number of *DoorBell* rings from n to 1.

Inlining. With this feature toggled on in a Verbs' WQE, the CPU copies the data into the WQE. Hence, with its first DMA read for the WQE, the NIC gets the payload as well, eliminating the second DMA read for the payload.

Unsignaled Completions. Instead of signaling a completion for each WQE, Verbs allows the application to turn off completions for WQEs provided that at least one out of every *n* WQEs is signaled. Turning off completions reduces the DMA writes of CQEs by the NIC. Additionally, the application polls fewer CQEs, reducing the overhead of making progress.

BlueFlame. *BlueFlame* is Nvidia-Mellanox's terminology for programmed I/O (PIO)—the CPU writes the WQE along with the *DoorBell*, eliminating the first DMA read. With *BlueFlame*, the UAR pages are mapped as write-combining (WC) memory (Intel terminology). Hence, the WQEs sent using *BlueFlame* are buffered through the CPU's WC buffers. Note that *BlueFlame* cannot be used with *Postlist*; the NIC must DMA-read the WQEs in the linked list. While Verbs itself does not provide an option to control PIO, device-specific methods exist to toggle PIO on or off.

Using both *Inlining* and *BlueFlame* for small messages eliminates two PCIe round-trip latencies. While the use of *Inlining* and *BlueFlame* is dependent on message size, the use of *Postlist* and *Unsignaled Completions* is reliant primarily on the user's design choices and application's semantics.

3.2 Communication Resources

To send messages across the network, the software (CPU) coordinates with the hardware (NIC) to *initiate* a transfer and confirm its *completion*. This coordination occurs through three communication resources: a software transmit queue, a software completion structure, and a NIC's hardware resource. The three interact using the mechanisms described in Section 2.1.3 and features described in Section 3.1.3. For Mellanox RDMA NICs, the transmit queue is the queue pair (QP), the completion structure is the completion queue (CQ), and the hardware resource is the uUAR contained within a UAR page (see Section 3.1.2). The QP, UAR, and uUAR make up the *initiation* interface; the CQ is the *completion* interface.

The communication from a core maps to QPs, and the QPs map to a uUAR on a UAR of the NIC. The interconnect's driver dictates the mapping between the transmit queues and the hardware resources while the user decides the mapping between the transmit queues and completion structures—multiple QPs could share the same CQ, or each could have its own.

The QP and CQ are associated with circular buffers that contain their work queue elements (WQEs) and completion queue entries (CQEs), respectively. The CPU writes to the QP's buffer, and the NIC DMA-reads it (when the *Inlining* feature is not used). The NIC DMA-writes the CQ's buffer and the CPU reads it when polling for progress. Both buffers are pinned by the operating system during resource creation.

The QP and CQ occupy memory with their circular buffers. So, every time we create a

Table 3.1: Memory (bytes) used by Verbs resources

CTXs	**PDs**	**MRs**	**QPs**	**CQs**	**Total**
256K	144	144	80K	9K	345K

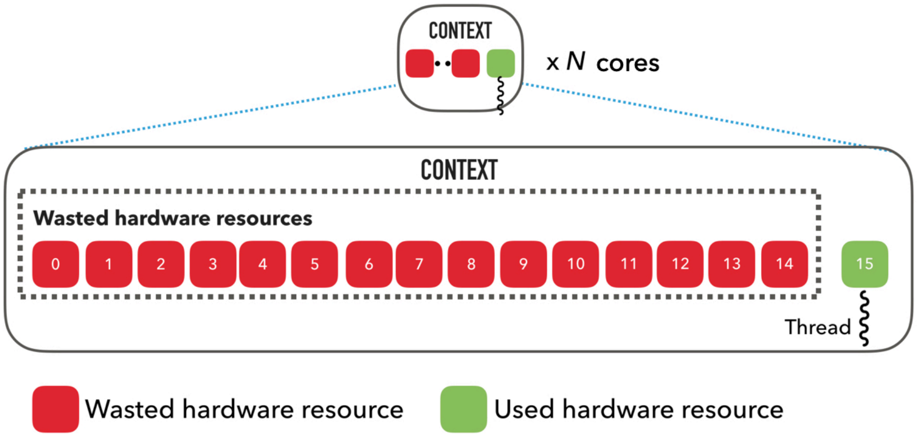


Figure 3.2: 93.75% hardware resource wastage in MPI everywhere.

QP or a CQ, we impact memory consumption. Table 3.1 shows the memory used by each type of a Verbs resource that is required to open a QP. Creating one communication portal requires at least 354 KB of memory, with the CTX occupying 74.2% of it.

However, the memory usage of the QP and the CQ is on the order of kilobytes, whereas the memory on the nodes of clusters and supercomputers is typically on the order of hundreds of gigabytes. Hence, we will notice a formidable impact on memory consumption only when the number of the Verbs resources is on the order of thousands. The impact of creating a QP or a CQ on memory is not of immediate concern.

On the other hand, the limit on the hardware resource is much smaller: 8K UAR pages on the ConnectX-4 NIC with only two uUARs per UAR. The situation is similar for the NICs of other interconnects such as Intel Omni-Path's Host Fabric Adapter (HFI) , which contains a maximum of 160 hardware contexts [7]. The 8K UARs on ConnectX-4 translates to a maximum of 1024 CTXs, given that each CTX allocates 8 UAR pages by default. This upper limit reflects the maximum number of communication channels a multiprocess environment (MPI everywhere) can create since each process needs its own context. Since each process uses only one uUAR out of the allocated 16 in a CTX, the resource wastage in MPI everywhere's network utilization is a staggering 93.75% (see Figure 3.2).

Arguably, even if a multithreaded environment were to create multiple communication channels by emulating those in MPI everywhere (a CTX per thread), it would not exhaust the NIC's hardware resources since the core count on modern processors is still far from 1024. The high resource wastage, however, is not a scalable approach. Eliminating this huge wastage would enable vendors to significantly reduce the power requirements and cost of their NICs. Multithreaded environments have the opportunity to prevent such high resource wastage since threads can utilize the multiple uUARs allocated to the CTX.

3.3 Resource-Sharing Opportunities

In a multithreaded environment (MPI+threads), a thread can map to the hardware resources in four possible ways. Figure 3.3 demonstrates the four ways described below.

1. *Maximum independence* – No sharing of any hardware resource between the threads; each maps to its own UAR page (same as in MPI everywhere).
2. *Shared UAR* – Threads map to distinct uUARs but share the same UAR page.
3. *Shared uUAR* – Threads use their own QPs, but the distinct QPs share the same uUAR. A lock is needed on the shared uUAR for parallel *BlueFlame* writes.
4. *Shared QP* – The threads share the same QP (same as in state-of-the-art MPI+threads), in which case a lock on the QP is needed for concurrent device WQE preparation. The lock on the QP also protects concurrent *BlueFlame* writes on the uUAR since the lock is released only after a *BlueFlame* write.

Sharing software and hardware communication resources at different levels improves resource efficiency but can hurt throughput. Next, we explore this tradeoff space between resource efficiency and communication throughput.

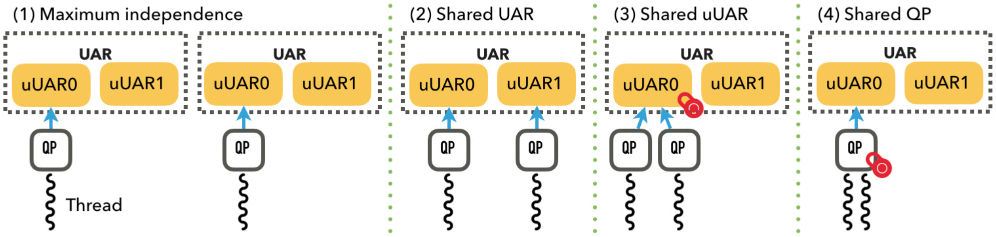


Figure 3.3: Four levels of thread-to-network-resource mapping in mlx5 between independent threads.

3.4 Experiment setup

To evaluate the impact of sharing resources on performance, we write a multithreaded RDMA-write message rate benchmark. We choose RDMA writes to eliminate any effects of receiver-side processing on the critical path.

We conduct our study on the Joint Laboratory for System Evaluation's Gomez cluster (each node has quad-socket Intel Haswell processors with 16 cores/socket and one hardware thread/core) using the rdma-core library [15]. Each node hosts a Mellanox ConnectX-4 NIC. We ensure that each thread is bound to its own core. For repeatable and reliable measurements, we disable the processor's turbo boost and set the CPU frequency to its base frequency: 2.5 GHz.

The design of our message-rate benchmark is adopted from `perftest` [12]. The loop of a thread iterates until all its messages are completed. In each iteration, the thread posts WQEs on a QP of depth, d in multiples of *Postlist*, p requesting for one signaled completion every q WQEs, where q is the value of *Unsignaled Completions*. In each poll on the CQ, the thread requests for $c = d/q$ completions, namely, all possible completions in an iteration. The depth of the CQ is c.

Postlist and *Unsignaled Completions* control the rate and amount of interaction between the CPU and NIC. Empirically, we find that setting $p = 32$ and $q = 64$ achieves the maximum

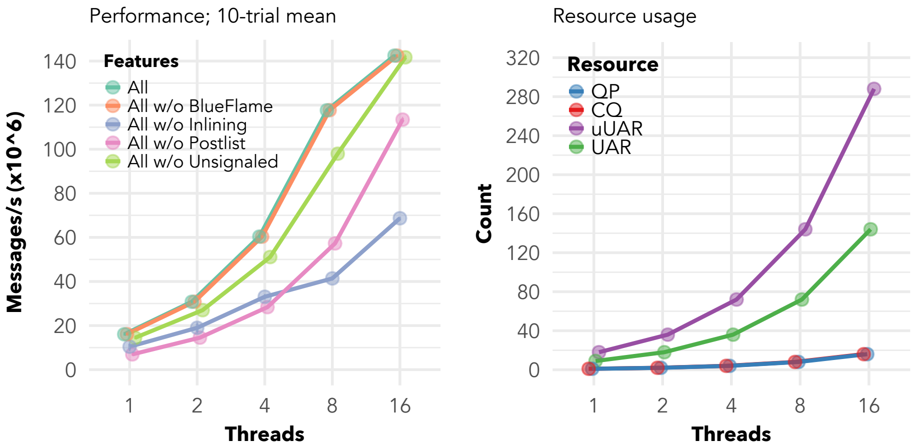


Figure 3.4: Performance (left) and resource usage (right) scalability with a CTX per thread.

throughput for 16 threads; hence, we use them as our default values. Note that we define the values of *Postlist* and *Unsignaled Completions* with respect to the threads, not to their associated QPs (relevant when threads share QPs).

To study the effect of an Verbs operational feature, we remove that feature while using others, referring to this case as "All w/o f," where f is the feature of interest. To disable *BlueFlame*, we set the MLX5_SHUT_UP_BF environment variable. To enable *Inlining*, we set the IBV_SEND_INLINE flag on the send-WQE. We use "w/o Postlist" to mean $p = 1$, and similarly "w/o Unsignaled" to mean $q = 1$.

Figure 3.4 shows the scalability[2] of communication throughput across the operational features of RDMA NICs, and it shows the communication resource usage of communication channels created with one context per thread for 2-byte RDMA writes. The number of QPs and CQs is an identity function of the number of threads; their memory consumption increases from 89 KB with one thread to 1.39 MB with 16 threads. The usage of UARs and uUARs also increases, but by a factor of 9 and 18, respectively, because each CTX containing one TD-assigned-QP allocates 9 UARs, and each UAR consist of two uUARs.

[2] The NIC is attached only to the first socket; cross-socket NIC behavior is out of the scope of this work.

3.5 Resource-Sharing Analysis

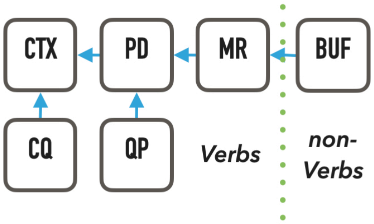


Figure 3.5: Hierarchical relation between the various Verbs objects (the arrow points to the parent); each object can have multiple children but only one parent.

A Verbs user allocates and interacts with the communication resources described in Section 3.1.3 through Verbs objects shown in Figure 3.5. Each of these objects represents a level of sharing between threads. Hence, we analyze the impact of sharing each Verbs object on performance and communication resource usage while considering the various operational features of RDMA NICs described in Section 3.1.3.

In the figures below, x-way sharing means the Verbs object of interest is being shared between x threads. For example, 8-way sharing for 16 threads means two instances of the object exist with 8 threads sharing one instance of the object. Moreover, we are interested in the change in throughput with increasing sharing rather than the absolute throughput obtained.

Starting with naïve communication channels—each thread using its own set of Verbs objects, that is, each thread using a dedicated Verbs CTX as in MPI everywhere—we move down each level of Verbs object sharing according to the hierarchical relation shown in Figure 3.5. Figure 3.4 shows the performance and communication resource usage of naïve communication channels for 16 threads.

3.5.1 Memory Buffer Sharing

The highest level of sharing is a non-Verbs object: the memory buffer. We define the BUF to be the pointer to the payload of the message. If the payload size is small enough, it can be inlined within the WQE; that is, the CPU will read it. By default, the maximum message size that can be inlined on ConnectX-4, exposed through Verbs, is 60 bytes; for any larger message size the NIC must DMA-read the payload.

Performance. When the CPU reads the payload, sharing this BUF between the threads is safe since parallel reads to the same memory location in a CPU are harmless. When the NIC reads the payload, however, its TLB design is important to consider (the DMA read entails a virtual-to-physical address translation). RDMA NICs typically employ a multi-rail TLB design to handle multiple transactions in parallel and sustain the high speed of the NIC's ASIC. The parallel load is distributed between the TLBs using a hash function. If this hash function is based on the cache line, parallel DMA reads to the same cache line will hit the same translation engine, serializing the reads. With a shared BUF the WQEs of multiple threads point to the same cache line, serializing the DMA reads.

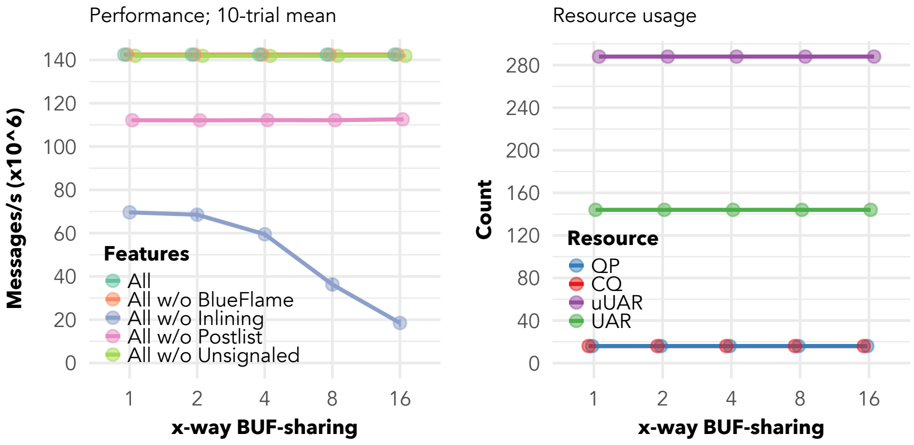


Figure 3.6: Message rate (left) and communication resource usage (right) with increasing BUF sharing across 16 threads.

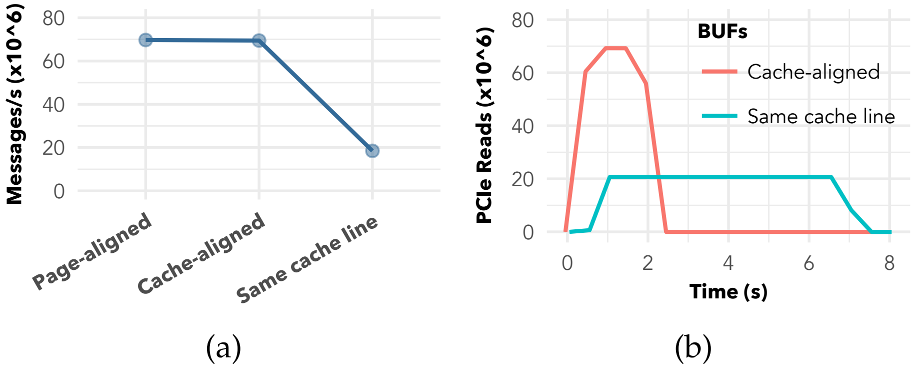


Figure 3.7: Effects on (a) message rate and (b) PCIe reads with and without cache-aligned buffers.

Figure 3.6 shows that the throughput indeed decreases with increasing BUF sharing when the NIC DMA-reads the payload (*i.e.* when messages are sent without *Inlining*). As validation of our analysis, Figure 3.7(a) shows that transmitting independent 2-byte buffers without 64-byte cache alignment hurts performance since all 16 buffers are on the same cache line. While the total numbers of PCIe reads (measured using PMU tools [14]) with and without cache alignment are equal, Figure 3.7(b) shows that the rate of the PCIe reads is much slower when the buffers are on the same cache line than when the buffers are cache aligned.

Resource usage. The BUF is a non-Verbs object. Hence, it does not affect the usage of any of the communication resources, as we can see in Figure 3.6.

3.5.2 Device Context Sharing

We note that the Verbs user gets maximally independent (level 1 in Figure 3.3) paths without CTX sharing since the QPs transparently get assigned to uUARs on different UARs. Within a shared CTX, however, the user, by default, has no way to explicitly request maximally independent paths for multiple QPs. When the user creates multiple thread domains (TDs), the mlx5 provider is currently hardcoded to assign the TD-assigned-QPs

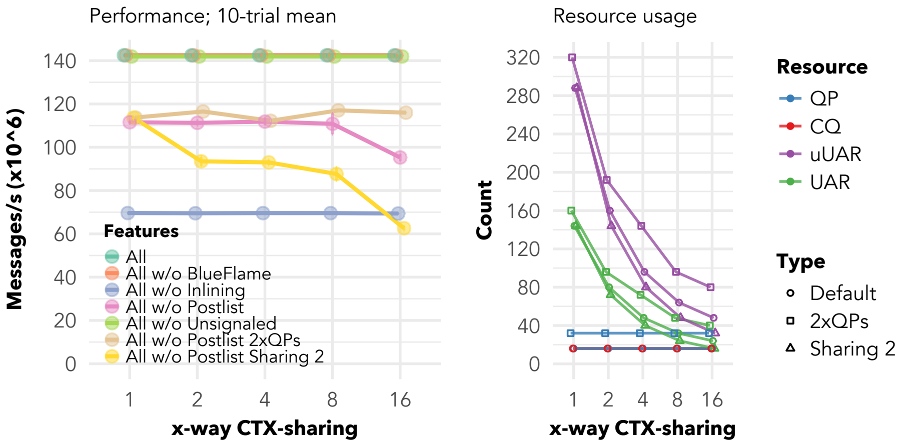


Figure 3.8: Message rate (left) and communication resource usage (right) with increasing CTX sharing across 16 threads.

to a uUAR using the second level of sharing, as shown in Figure 3.3. More abstractly, Verbs users today have no way to request a sharing level for the QPs they create.

To overcome this Verbs design limitation, we propose a variable, `sharing`, in the TD initialization attributes (`struct ibv_td_init_attr`) that are passed during TD creation. The higher the value of `sharing`, the higher is the amount of hardware resource sharing between multiple TDs. A `sharing` value of 1 refers to maximally independent paths. In mlx5, only two levels of sharing exist for TDs, corresponding to (1) and (2) in Figure 3.3.

Performance. For maximally independent threads, sharing the CTX should not affect performance since we emulate the thread-to-uUAR mapping in the MPI everywhere model. Sharing a CTX with the second level of sharing between threads could hurt performance since the uUARs on the same UAR could share the same set of the NIC's registers. Additionally, the CPU architecture's implementation of flushing write combining memory can impact performance in the second level of sharing since the memory attribute of the uUARs is set at the page-level granularity by using the Page Attribute Table (PAT) [13].

Figure 3.8 shows that sharing the CTX does not hurt performance except when we do not use *Postlist* that is, when we use *BlueFlame* writes (programmed I/O). For example, we notice a 1.15x drop in performance going from 8-way to 16-way CTX sharing even with maximally independent QPs. While the engineers at Mellanox were able to reproduce this drop even on a newer ConnectX-5 adapter, the cause for the drop is unknown. We discovered that creating twice the number of maximally independent QPs but using only half of them (even or odd ones) can eliminate this drop, as seen in the "All w/o Postlist 2xQPs" line. Additionally, from the "All w/o Postlist Sharing 2" line, we can see the harmful effects of sharing a UAR when the mlx5 provider is hardcoded to use the second sharing level for assigning TDs within a shared CTX to uUARs.

While this evaluation validates the need for maximally independent paths, it does not explain the decline in throughput when there are concurrent *BlueFlame* writes to distinct uUARs sharing the same UAR page. Finding the precise reason for this behavior is hard since the hardware-software interaction is dependent on proprietary technologies.

Resource usage. Sharing the CTX is critical for hardware resource usage, as seen in Figure 3.8. The reason is that a maximally independent TD-assigned-QP within a shared CTX adds only 1 UAR as opposed to the 9 UARs that are allocated when a QP is created with a new CTX. Also, the 16 uUARs and 8 UARs statically allocated by the mlx5 provider during CTX creation (see Section 3.1.2) are wasted only once. Nonetheless, maximally independent QPs waste one uUAR per thread. While sharing the CTX does not impact QP and CQ usage, it does reduce the overall memory consumption. For example, when shared between 16 threads, the overall memory consumption reduces by 9x (from 5.15 MB to 0.35 MB).

Creating twice as many TD-assigned-QPs ("2xQPs" in Figure 3.8) increases resource usage since each of the extra 16 maximally independent QPs allocates their own QP and UAR. The second level of sharing that mlx5 is hardcoded to use consumes 2x fewer UARs than do maximally independent QPs.

3.5.3 Protection Domain Sharing

The protection domain is just a means of isolating a collection of Verbs objects. Objects contained under different PDs cannot interact with each other.

Performance. The software PD object is not accessed on the critical data-path; the protection checks occur in the NIC. Hence, from a performance perspective, sharing a PD between multiple threads is harmless, as observed in Figure 3.9.

Resource usage. The PD does not impact the usage of any of the communication resources, as we can see in Figure 3.9. The uUAR and UAR values reflect those of one CTX since the PD can be shared only within a CTX.

3.5.4 Memory Region Sharing

The MR object pins memory in the virtual address space of the user. The operating system prepares the pinned memory for DMA accesses from the NIC.

Performance. Sharing the MR between threads has no impact on performance since the

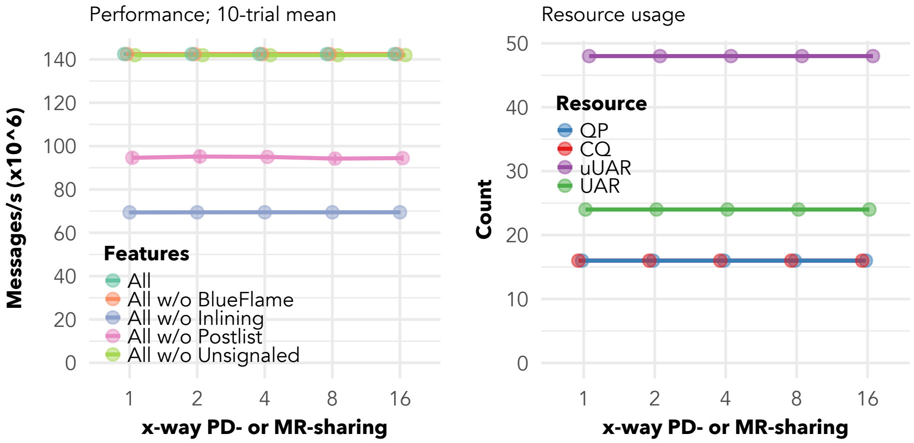


Figure 3.9: Message rate (left) and communication resource usage (right) with increasing PD or MR sharing across 16 threads.

MR is just an object that points to a registered memory region. The MR may span multiple contiguous BUFs. Sharing an MR containing only one BUF means that the threads are sharing the BUF as well, which implies the same effects of BUF sharing. Figure 3.9 confirms that sharing the MR does not affect performance as long as the threads have independent cache-aligned buffers.

Resource usage. The MR does not control the allocation of any of the communication resources. Hence, sharing it will have no impact, as we can see in Figure 3.9.

3.5.5 Completion Queue Sharing

The Verbs user can map multiple QPs to the same CQ, allowing for CQ-sharing between threads. In HPC scenarios, the user actively polls the CQ (as opposed to relying on signals involving the kernel) on the critical data-path to confirm progress in communication.

Performance. The CQ has a lock that a thread acquires before polling it. Hence, the threads sharing a CQ contend on the CQ's lock. Additionally, if QP i and QP j share a CQ, then thread i driving QP i can read QP j's completions. Hence, the completion

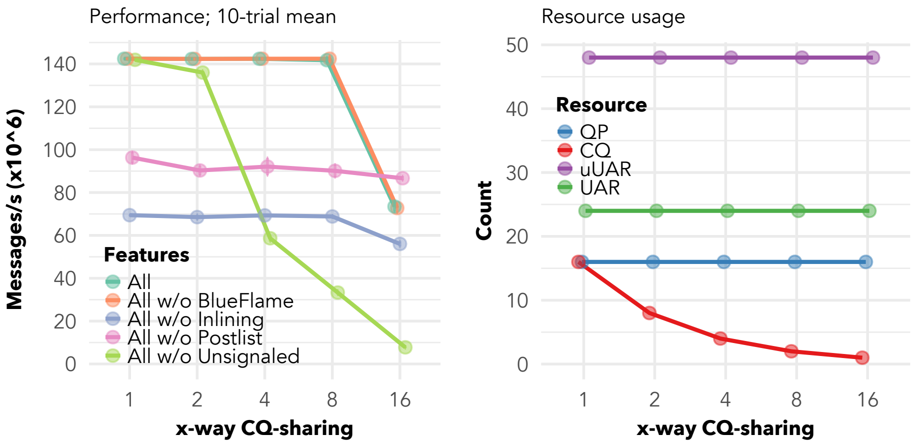


Figure 3.10: Message rate (left) and communication resource usage (right) with increasing CQ sharing across 16 threads.

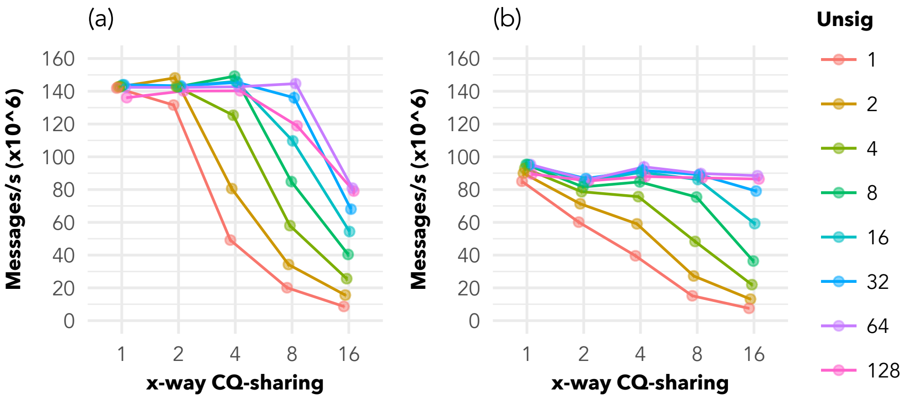


Figure 3.11: (a) Postlist size of 32, (b) Postlist size of 1.

counter for any thread k requires atomic updates. Atomics and locks are obvious sources of contention when sharing CQs between threads. Figure 3.10 demonstrates these hurtful effects of CQ sharing. The effects are most noticeable in 16-way sharing because there exists a tradeoff space between the benefits of *Unsignaled Completions* and the overheads of CQ sharing. Figure 3.11(a) portrays this tradeoff space. Lower values of *Unsignaled Completions* imply that the thread reads more completions from the CQ than for higher values, translating to a longer hold-time of the shared CQ's lock. Thus, the impact of lock contention is most visible in "All w/o Unsignaled." For higher *Unsignaled Completion*-values, we see a drop only after a certain level of CQ sharing because the benefits of *Postlist* outweigh the impact of contention. Removing *Postlist* shows a linear decrease in throughput with increasing contention in Figure 3.11(b).

We note that even if the Verbs user can guarantee single-thread access to a CQ, the standard CQ does not allow the user to disable the lock on the CQ. The extended CQ, on the other hand, allows the user to do so during CQ creation (`ibv_create_cq_ex`) with the `IBV_CREATE_CQ_ATTR_SINGLE_THREADED` flag.

Resource usage. Sharing the CQ translates to fewer circular buffers, and hence it reduces the memory consumption of the completion communication resource. But it does not affect hardware resource usage, as we can see in Figure 3.10. The uUAR and UAR usage shown corresponds to that of one CTX since a CQ can be shared only within a CTX.

3.5.6 Queue Pair Sharing

Ultimately, the user can choose to share the queue pair between threads to achieve maximum resource efficiency. This is the case in state-of-the-art MPI implementations.

Performance. The QP has a lock that a thread needs to acquire before posting on it. Hence, threads contend on a shared QP's lock. Additionally, the threads need to coordinate to post on the finite QP-depth of the shared QP using atomic fetch-and-decrement operations on the QP-depth value. These locks and atomics are sources of contention when sharing QPs. Most important, the NIC's parallel network resources are not utilized with shared QPs since each QP is assigned to only one hardware resource through which the messages of multiple threads are serialized. Figure 3.12 shows the expected decline in throughput with increasing QP-sharing. Removing *Postlist* is more detrimental than removing *Unsignaled Completion* because the contention on the QP's lock without *Postlist* is

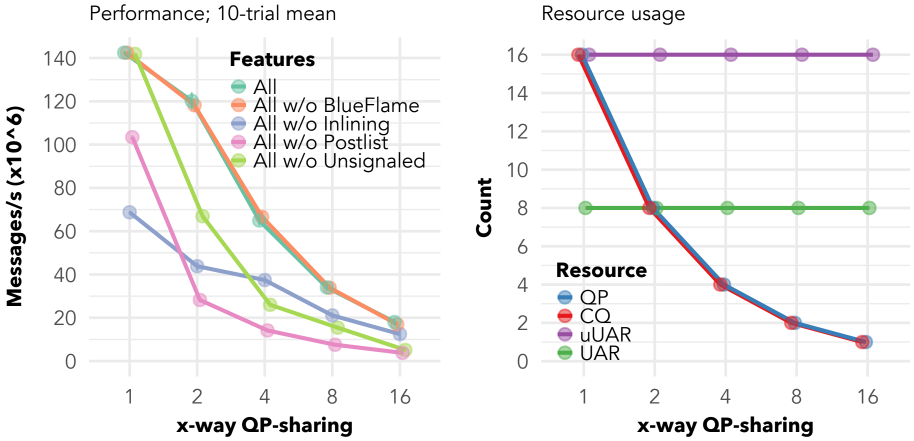


Figure 3.12: Message rate (left) and communication resource usage (right) with increasing QP sharing across 16 threads.

higher.

Resource usage. Sharing the QP means fewer circular buffers for the WQEs and hence lower memory consumption. It does not affect hardware resource usage, as we can see in Figure 3.12. QP sharing reduces the number of both QPs and CQs, reducing the total memory consumption of the software communication resources by 16x with 16-way sharing.

3.5.7 Summary of Lessons Learned

Our resource-sharing analysis in this section showcases the tradeoff space between communication performance and resource usage. Below is a summary of the key lessons learned from our analysis.

- Each thread must have its own cache-aligned buffer to prevent a performance drop.
- CTX-sharing is the most critical for the usage of hardware resources. With 16-way sharing, “2xQPs” can achieve the same performance as independent CTXs using 80 uUARs instead of 288. If 20% less performance is acceptable, we can use maximally independent TDs that use 6x fewer resources. If 50% less performance is acceptable, we can use “Sharing 2” that uses 9x fewer resources.
- Sharing the PD or the MR will not hurt performance, while keeping them independent will not utilize any communication resource.
- Only QP- and CQ-sharing affects the memory consumption of the software resources. However, the reduction in memory usage by sharing them is not as critical as the consequent drop in performance. For example, 16-way sharing of the CQ improves memory usage by 1.1x but can result in an 18x drop in performance.

3.6 Resource-Sharing Model

Building on our analysis, we define a resource sharing model—*scalable communication channels*—that concretely categorizes the design space of creating multiple communication channels into six categories. Below we describe the design of the initiation interface in each category, state how the user can create it, discuss what occurs internally in the RDMA software stack, and discuss its implications on performance and resource usage. In all the categories, we maintain a separate CQ for each QP.

MPI everywhere. This category emulates the endpoint configuration when multiple ranks run on a node. It represents level 1 in Figure 3.3. The user creates this by creating a separate CTX for each thread, each containing its own TD-assigned-QP and CQ. Within each CTX, the mlx5 driver assigns the QP to a uUAR. Since each CTX contains 9 UARs (8 statically allocated and 1 dynamically allocated with the thread domain), consecutive QPs transparently get assigned to distinct UAR pages. The performance of this category is best possible since there is no sharing of resources. The resource usage of this category, on the other hand, is high. Additionally, it is wasteful since only 1 of the 18 allocated uUARs is used per thread. The memory consumption increases linearly with the number of threads since the number of QPs and CQs is an identity function of the number of threads.

2xDynamic. This category also represents a 1-to-1 mapping between a uUAR and a thread. Unlike ″MPI everywhere,″ however, the user creates only one CTX for all the threads and creates twice as many TD-assigned-QPs as threads. The threads use only the even or odd QPs. The mlx5 provider dynamically allocates a new UAR page for each TD-assigned-QP and assigns the first uUAR to the QP, enabling a 1-to-1 mapping. Like ″MPI everywhere,″ this category also delivers the best performance. Since the number of QPs is twice the number of threads, however, each thread wastes 1 dynamically allocated UAR, 3 uUARs, and 1 QP. The memory consumption of QPs and CQs is twice that of MPI everywhere. The statically allocated hardware resources are wasted regardless of the number of threads.

Dynamic. This category also represents a 1-to-1 mapping between a uUAR and a thread, but the number QPs equals the number of threads. The user creates this configuration similarly to that in ″2xDynamic″ but creates only as many QPs as threads. According to Section 3.5.2, this configuration hurts communication throughput. In terms of resource usage, however, only one uUAR is wasted per thread. The 8 statically allocated UARs are naturally wasted; none of the dynamically allocated UARs are wasted. The memory consumption of QPs and CQs is half of that in "2xDynamic″ and same as that in ″MPI everywhere″.

Shared Dynamic. This category represents level 2 in Figure 3.3. The user creates this configuration using a shared CTX, similar to the way in "Dynamic,″ but assigns each QP to a TD with the second level of sharing. The mlx5 driver will dynamically allocate UARs only for the even QPs and map the even QPs to the first uUAR and the odd QPs to the second uUAR of the allocated UAR. According to Section 3.5.2, sharing the UAR will hurt performance. The hardware resource usage is less than with "Dynamic″ since only half as many UARs and uUARs as threads are allocated. Apart from the 8 statically allocated UARs and uUARs, none of the dynamically allocated resources are wasted. The memory consumption of QPs and CQs is equivalent to that of "Dynamic.″

Static. The user uses the statically allocated resources within a CTX, resulting in a many-to-one mapping between the threads and uUARs (and UARs). To do so, the user simply creates a QP for each thread within a shared CTX without any TDs. The final state of the mapping for a given number of QPs is dependent on the driver's assignment policy. In mlx5, with 16 QPs, we end up with a combination of the second and third level of sharing in Figure 3.3—the 5th and 16th QP are mapped to the same uUAR (third level), while the others are mapped to the rest of the uUARs using the second level of sharing. The hardware resource usage is the number of statically allocated resources. Resources are wasted only when the number of threads is less than 16. The memory consumption is equivalent to that of “Dynamic.”

MPI+threads. This category represents level 4 in Figure 3.3. The user creates this by creating only 1 CTX, 1 QP, and 1 CQ. The mlx5 driver assigns the one QP to a low-latency uUAR. The performance of this category is the worst possible since the communication of all the threads is bottlenecked through one QP. The resource usage of this category is not a function of the number of threads and hence is the best possible. All threads allocate only 8 UARs, 16 UARs, 1 QP, and 1 CQ.

Note that the CQ can be shared in any manner in the above categories and its impact is orthogonal to the effects of the initiation interface.

3.6.1 Demonstration

We showcase the scalable communication channels resource-sharing model described in Section 3.6 on a global arrays communication kernel as an example using our two-node evaluation setup. We could not use more than two nodes since thread domains are supported only on Linux kernel 4.16 onward, and the latest stable kernel at the time of experiments was 4.17.2; the combination of a mlx5 device along with the latest stable kernel was rare. We limit our evaluation to the semantics used in MPI libraries, that is, we do not use the *Postlist* and *Unsignaled Completions* features; we use *BlueFlame* since it is toggled on by default.

The pattern of fetching and writing tiles from and to a global array is at the core of many scientific applications such as NWChem [101], which constitutes a multidimen-

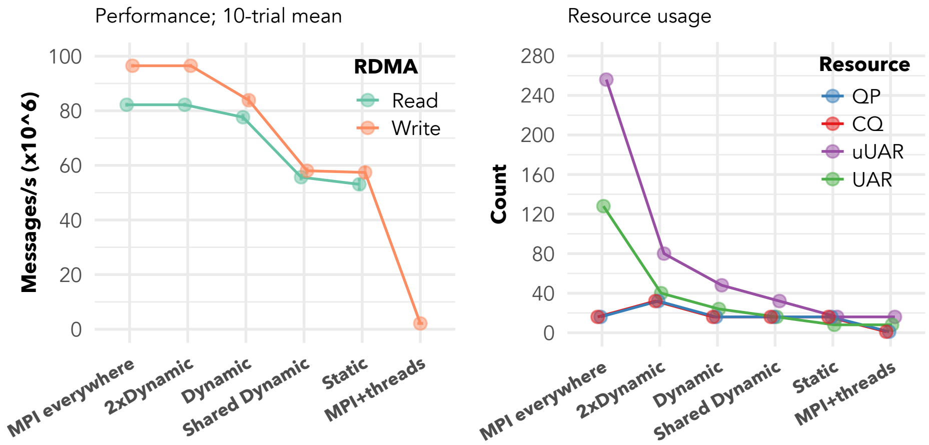


Figure 3.13: Scalable communication channels for the global array kernel with 16 threads. Left: Communication throughput. Right: Communication resource usage

Table 3.2: Scalable Communication Channels for Global Arrays.

Category	Performance	Hardware resources	Software resources
MPI everywhere	100%	100%	100%
2xDynamic	100%	31.25%	200%
Dynamic	94%	18.75%	100%
Shared Dynamic	65%	12.5%	100%
Static	64%	6.25%	100%
MPI+threads	3%	6.25%	6.25%

sional double-precision matrix multiply (DGEMM). We implement a DGEMM benchmark ($A \times B = C$), where the global matrices A, B, and C reside on a server node and a client node performs the DGEMM using Verbs for internode communication. We design the benchmark such that all the QPs share the same PD but each has three BUFs and three MRs—one for each of the three tiles from A, B, and C.

Figure 3.13 shows the performance and resource usage of scalable communication endpoints for 16 threads. We can observe that performance decreases with lower resource usage. Table 3.2 shows the performance and resource usage of the different categories normalized to those of the MPI everywhere category which achieves the highest performance and highest resource usage. The main takeaway here is that resource sharing in the 2xDynamic category enables multiple threads to achieve the same performance as having dedicated communication channels by using only 31.25% as many hardware resources.

The memory consumption of QPs and CQs is the same for all categories except 2xDynamic and MPI+threads. While the number of QPs and CQs in 2xDynamic is twice that of MPI everywhere, the overall memory usage in the former is 3.27x lower (1.64 MB vs 5.39 MB) since MPI everywhere has 16 CTXs while 2xDynamic has only one. The memory consumption is the lowest in MPI+threads with only one QP and one CQ.

3.7 Concluding Remarks

State-of-the-art MPI implementations either achieve maximum communication throughput and waste 93.75% of network hardware resources in MPI everywhere or achieve maximum resource efficiency but perform up to 7x worse in MPI+threads. Our study in this chapter shows that that a tradeoff space between communication performance and resource efficiency exists in regards to creating multiple communication channels in a multithreaded environment; the state of the art represents the two extreme ends of this tradeoff space. The scalable communication channels resource-sharing model categorizes this tradeoff space into six categories and shows that it is possible for a multithreaded environment to achieve the same performance as a multiprocess environment using just a third of the resources.

We conducted this study at the lowest Verbs level of the HPC software stack for full control during the resource-sharing analysis. Modern MPI libraries, however, use low-level communication frameworks such as UCX or OFI for their network operations to transparently obtain portability across the various interconnects. We can translate the lessons learned from our Verbs-object sharing analysis to the newer abstract frameworks. In UCX, for example, sharing the UCP Context between threads achieves the same network-

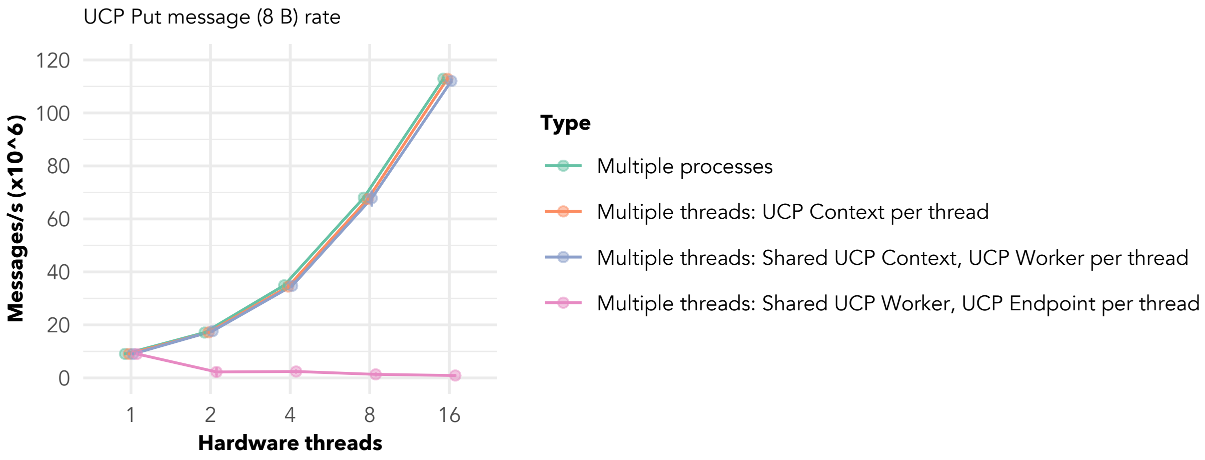


Figure 3.14: Performance of different levels of resource sharing in UCX.

hardware-resource savings that we witnessed by sharing the Verbs Context. Within a shared UCP Context, we can achieve the same performance as multiple processes do by using a distinct UCP Worker per thread as seen in Figure 3.14.

Chapter 4

A Fast MPI+Threads Library

Having learned that the modern network hardware enables scalable multithreaded communication both in terms of performance and resource usage, we move up a layer in the HPC software stack to create multiple communication channels inside the MPI library. State-of-the-art MPI libraries employ conservative semantics—a global critical section for thread safety and utilization of only a single network context per process—to maintain MPI's ordering constraints. The existing MPI standard, however, does provide ways to overcome its ordering constraints. Nonetheless, even if the application informs the MPI library that two or more messages have no relative order between them by using, for example, separate communicators for these messages, MPI libraries today do not funnel them through distinct network contexts.

Creating multiple network communication channels inside the MPI library is a challenging endeavor. On top of the challenge of managing the limited network resources, MPI libraries also need to ensure correct MPI semantics when funneling messages through multiple network contexts. For example, certain corner cases with respect to communication progress must be handled for correctness, even though they sometimes hurt performance. Most important, MPI libraries must ensure threads can access its software components such as the progress engine for parallel communication without contention; otherwise, mapping to multiple network resources is a futile effort.

In this chapter, we first describe the different ways in which a user can express logical communication parallelism with the existing MPI-3.1 standard. We then establish parallel communication streams inside the MPI library through fine-grained critical sections and virtual communication interfaces (VCIs) to execute logically parallel communication without any contention. A VCI represents a dedicated communication stream that is mapped to a distinct network hardware context. Most important, threads mapped to different VCIs do not contend with each other on the library's software resources. Unlike related work, correctness is a first-class citizen in our implementation, that is, we do not sacrifice correctness for performance. Our work is the first to achieve scaling MPI+threads communication performance that matches MPI everywhere communication performance for both point-to-point and RMA operations.

4.1 Parallelism in the Current MPI-3.1 Standard

First, we discuss the opportunities that the existing MPI-3.1 standard provides with respect to exposing logially parallel communication for point-to-point and RMA operations.

4.1.1 Point-to-Point Communication

For two-sided communication, MPI uses the $\langle$communicator, rank, tag$\rangle$ triplet to match operations.

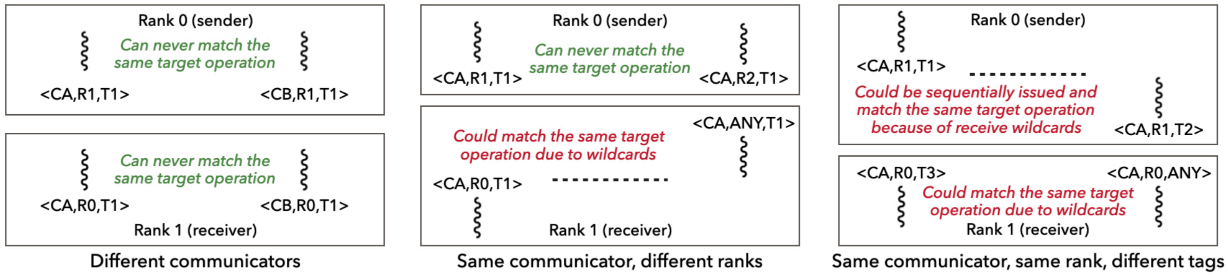

Rank 0 (sender)
Can never match the same target operation
<CA,R1,T1>
<CB,R1,T1>
Can never match the same target operation
<CA,R0,T1>
<CB,R0,T1>
Rank 1 (receiver)
Different communicators
Rank 0 (sender)
Can never match the same target operation
<CA,R1,T1>
<CA,R2,T1>
<CA,ANY,T1>
Could match the same target operation due to wildcards
<CA,R0,T1>
Rank 1 (receiver)
Same communicator, different ranks
Rank 0 (sender)
<CA,R1,T1>
Could be sequentially issued and match the same target operation because of receive wildcards
<CA,R1,T2>
<CA,R0,T3>
<CA,R0,ANY>
Could match the same target operation due to wildcards
Rank 1 (receiver)
Same communicator, same rank, different tags

Figure 4.1: Different combinations of ⟨comm,rank,tag⟩ tuples demonstrating point-to-point parallelism in MPI-3.1. Dashed horizontal lines represent thread barriers.

Different communicators. MPI does not define any order between operations executed on different communicators. This semantic implies that all operations on different communicators are logically parallel.

Same communicator, different ranks. Within a communicator, MPI specifies a *nonovertaking order* [16]: if multiple ordered operations match the same target operation, the operation that was issued first must consume the target operation before the one that was issued later. No matching order applies to operations intended for different targets. For example, no ordering constraints apply to multiple send operations that use the same communicator but target different ranks. Hence, they can execute on parallel communication streams. On the other hand, receive operations that use the same communicator cannot execute in parallel even if they specify different ranks. The reason is that it is possible for any receive operation to contain the `MPI_ANY_SOURCE` wildcard. To ensure correct matching order, the MPI library is forced to funnel all receive operations of a communicator through the same receive queue (see Figure 4.1).

Same communicator, same rank, different tags. Both send and receive operations that target the same rank within a communicator but use different tags are not logically parallel. The order of operations in MPI is determined by the MPI user. In MPI+threads, the MPI library cannot assume that operations on different threads occur in parallel; the operations may be ordered through, for example, a thread barrier. Suppose the user issues two operations on two different threads with a barrier between the operations (see Figure 4.1). A target operation that satisfies both operations must first match the operation that was issued before the barrier. To ensure this, the MPI library must use the same communication queue for the operations from the two threads. If the operations use different queues, the operation issued after the barrier could incorrectly match the target prior to the one issued before the barrier.

Summary. The discussion above explains why domain scientists can maximize the mulithreaded communication independence of point-to-point operations only through communicators. Using tags and ranks is not sufficient because of wildcards on the receive side.

4.1.2 RMA

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

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

4.2 Software and Testbeds

Our work is based on the highly optimized CH4 [87] device of the MPICH library, which, along with its derivatives, is the most widely used MPI implementation. The new CH4 device is a combination of three components: a core (`ch4_core`), a network module (`netmod`), and a shared-memory module (`shmmod`). The `netmod` and `shmmod` are responsible for conducting internode and intranode communication, respectively. In this work, we focus on the `netmod` component because MPI+threads applications would directly use the shared memory of the process for intranode communication.

For most common data operations, CH4 offloads functionalities, such as tag matching, to the low-level communication library, such as OpenFabrics Interfaces (OFI) [52] or Unified Communication X (UCX) [93]. Where the hardware cannot independently handle operations, CH4 falls back on using an active message implementation of the operation in its `ch4_core`.

Our testbeds include two platforms: the Skylake cluster and the Gomez cluster in the Joint Laboratory for System Evaluation at Argonne National Laboratory. The clusters feature different interconnects: Skylake hosts Intel Omni-Path (OPA) and Gomez hosts

Mellanox InfiniBand (IB) EDR. These two families of interconnects constitute the majority of the interconnect performance share on the TOP500 list [20] (as of the November 2020 rankings). For Skylake, we use the OFI `netmod` in conjunction with PSM2; for Gomez, we use the UCX `netmod` with Verbs.

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

4.3 Deserializing Access to the MPI Library

The critical precursor to utilizing network parallelism is fast multithreaded access to the MPI library itself. The global critical section in today's MPI implementations serializes communication from multiple threads even if the communication operations issued by those threads are independent. The MPI operation enters the critical section at the beginning of its execution and exits it either when it returns from the function or when it yields to other threads to make communication progress.

Fine-grained critical sections. Balaji et al. [31, 32] and Amer et al. [26, 27] split the global lock in MPICH into multiple locks such that each lock protects a different class of objects. For example, access to the network communication portal is protected by a lock different from the one that protects the management of request objects. Although fine-grained critical sections (FG) mean higher parallelism, they incur two expenses over a global critical section (Global): (1) more lock acquisitions and releases on the critical path and (2) atomics for reference and completion counters.

The number of locks taken in FG depends on the type of operations. For any initiation operation, we need at least one lock—the one that protects access to the communica-

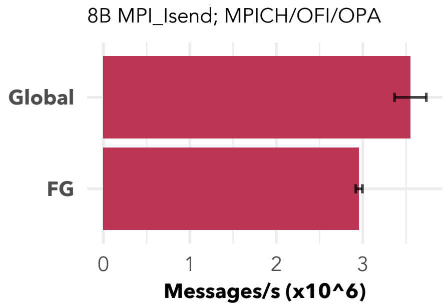


Figure 4.2: Overhead of FG.

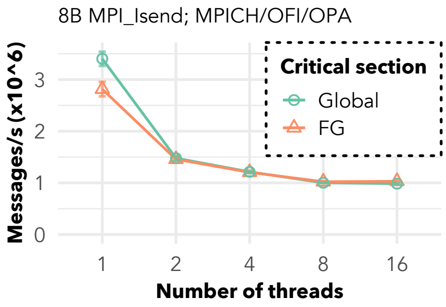


Figure 4.3: Global vs. FG.

tion portal. Generally, for `MPI_Isend` and `MPI_Irecv`, we need a second lock—the one that allocates a request object from the global pool of requests. For progress operations, the number of locks taken depends on the number of times the progress engine is invoked. The progress engine not only checks for the completion of an operation but also progresses active outstanding schedules, such as those of non-blocking collectives. One iteration of the progress engine in MPICH takes three locks: one to poll the communication portal and two to check the activeness of progress hooks[1] (each hook maintains its own thread safety). When an operation completes, another lock is taken when the request object is returned to the pool.

Although FG enables parallelism when multiple threads compete for MPI resources, it adds some overhead when there is no contention (e.g., when a single thread is active). Figure 4.2 shows that, compared to Global, FG hurts performance by 16.71% in the uncontended case. This performance difference is due to the higher number of locks and to atomic counting (as we corroborate in Section 4.5.3). With increasing number of threads, the performance difference between FG and Global reduces, and FG eventually outperforms Global at 16 threads, as seen in Figure 4.3. Moreover, although Global performs better than FG for fewer threads, FG is critical when parallel communication streams exist even in uncontended case, as we show in Section 4.4.

[1] MPICH/CH4 currently maintains two progress hooks.

4.4 Fast Parallel Communication Streams

To address the problem of network resource underutilization, we first define the virtual communication interface (VCI) object. A VCI is an abstract representation of a communication stream. Each VCI maps to a communication context on the network hardware and contains its own independent set of communication resources that maintain a FIFO order of the MPI operations that map to it. Hence, with multiple VCIs, MPI libraries can obtain parallel communication streams. The physical realization of a VCI depends on the `netmod` and the underlying interconnect. A VCI in the OFI `netmod` is an OFI endpoint (for transmission and reception) that is bound to an OFI completion queue (for progress). For Intel OPA, the OFI endpoint maps to a hardware context on the Intel HFI network adapter [7]. 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 (see Appendix A).

Thread safety. We extend the fine-grained critical sections from Section 4.3 such that each VCI is associated with a distinct lock since each VCI is independent. The resources of a VCI are then protected by the lock of the VCI. Threads that map to different VCIs can access the VCIs without contention.

Connection establishment. Each VCI has its own transport-level address that needs to be exchanged between the ranks in order to establish connections. We do so during the initialization of MPI. We first use PMI [30] to exchange the addresses of the zeroth VCIs on every rank. Using the zeroth VCI, we exchange the addresses of the rest of the VCIs using an allgather operation. As expected, establishing connections statically during initialization incurs an overhead that grows with the number of VCIs (see Figure 4.4). Similarly, the finalization time increases since the tear-down time of VCIs is proportional to the

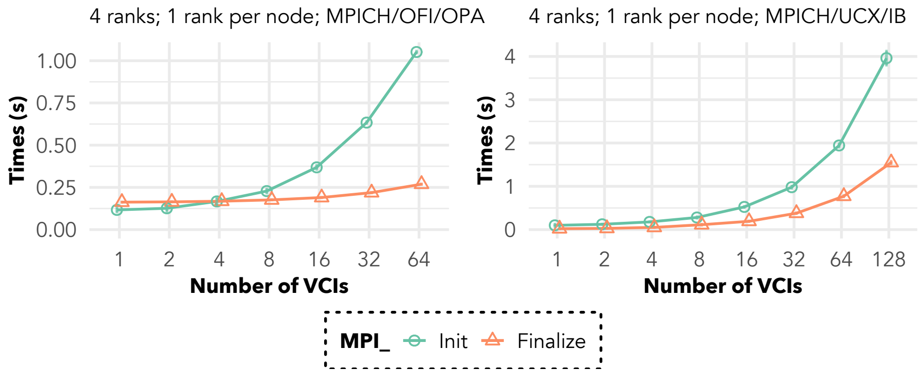


Figure 4.4: Multi-VCI `MPI_Init` and `MPI_Finalize` overheads.

number of VCIs.[2]

Mapping logical communication parallelism to network parallelism. Once an MPI+threads application exposes the logical parallelism in its communication, MPI libraries can map it to their internal VCIs so that logically parallel communication can funnel through distinct network contexts. As we learned in Section 4.1, communicators express parallelism between all point-to-point operations. But since users may use multiple communicators for purposes apart from exposing logically parallel communication, we allow users to request a new VCI for a communicator through an Info hint[3]. By default, if the user does not supply any Info hint, all new communicators point to the same single VCI. The same approach applies to the creation of new windows since all RMA operations on different windows are unordered. With RMA operations, however, users have the additional option of letting the MPI library automatically map the multithreaded RMA operations within a window since some of them (e.g., `MPI_Get`) can be unordered. For such cases, we allow the user to request multiple VCIs for a window through Info hints. Then, one way to automatically map multithreaded RMA operations to the mul-

[2] Features like OFI scalable endpoints can reduce the connection establishment and tear-down overheads, because they share the same transport-level address. However, we have not used them in this work because their performance is still not on par with that of regular endpoints, at least for the PSM2 provider that we used in this work. Furthermore, scalable endpoints share some resources, such as the OFI address vector, accesses to which could be serialized in the critical path by the OFI provider [17].

[3] Introducing new implementation-specific Info hints does not violate the MPI-3.1 standard.

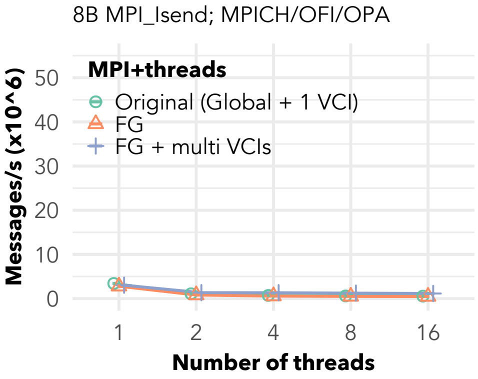


Figure 4.5: Fine-grained critical sections with multiple VCIs.

tiple VCIs allocated to the window is through a hash function whose key is the thread ID. This approach, however, suffers primarily from collisions in the hash function. Users can benefit from automatic mapping when the semantics of MPI prevent them from using multiple windows but the application does not rely on ordering of operations (e.g., multithreaded `MPI_Accumulate` in NWChem as we detail in Section 4.6.3).

Fallback mechanisms. As we learned in Chapter 3, however, the number of network hardware resources on a NIC is limited. So, it is likely for the VCI pool to be empty during communicator or window creation. For such cases, MPI libraries must maintain fallback policies while mapping the application's logically parallel communication to the network hardware contexts. One example of such a fallback policy would be to round-robin over the already allocated VCIs.

Simply employing multiple VCIs in an MPI library with fine-grained critical sections, however, yields no performance benefit (see Figure 4.5). Together they perform similarly to the original version of the MPI library which employs a global critical section and uses a single VCI.

4.4.1 Optimizing multi-VCI communication

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 dissolve bottlenecks present in the architectures of MPI libraries. Although our work has been on MPICH, the concepts apply to other MPI libraries as well.

Per-VCI Progress

With only one VCI (Original), the job of the progress function was simple: poll for progress on the single VCI. With multiple VCIs, a naïve extension would be to poll for progress on all the active VCIs. Although correct, this approach would be detrimental to performance especially when multiple threads progress operations in parallel since they would contend on the VCIs' locks. Also, each thread would be doing more work than necessary. Because all MPI communication operations map to a VCI, progress for an operation primarily needs to poll the VCI on which the operation was posted. We extend the progress engine to allow for *per-VCI progress*. First, we store the VCI used for an operation in its request object. Using the information stored in the request object, the progress functions poll for progress on the VCI that was used for the operation. When multiple threads progress operations mapped to different VCIs, they do not contend.

Although per-VCI progress helps improve performance, progressing only the VCI used by the current request is incorrect and can lead to deadlock. Consider the point-to-point example in Figure 4.6. This is a correct MPI program—the first synchronous send[4] on rank 0 (line 3) should return because its matching receive has already been posted (line 9). With current MPI libraries, this program completes because `MPI_Wait(req2)` (line 17)

[4] conceptually similar to an `MPI_Send` following the rendezvous protocol.

```
/*Point-to-point example*/
Rank 0:
  MPI_Ssend(comm1);
  MPI_Ssend(comm2);



Rank 1 / Thread 0:
  MPI_Irecv(comm1,req1);
#pragma omp barrier
#pragma omp barrier
  MPI_Wait(req1);

Rank 1 / Thread 1:
  MPI_Irecv(comm2,req2);
#pragma omp barrier
  MPI_Wait(req2);
#pragma omp barrier
```

```
/*RMA example (large Puts)*/
Rank 0:
  MPI_Get(win1);
  MPI_Get(win2);
  MPI_Win_flush(win1);
  MPI_Win_flush(win2);

Rank 1 / Thread 0:
  MPI_Get(win1);
#pragma omp barrier
#pragma omp barrier
  MPI_Win_flush(win1);

Rank 1 / Thread 1:
  MPI_Get(win2);
#pragma omp barrier
  MPI_Win_flush(win2);
#pragma omp barrier
```

Figure 4.6: Point-to-point (left) and RMA (right) scenarios that would deadlock without shared progress of VCIs.

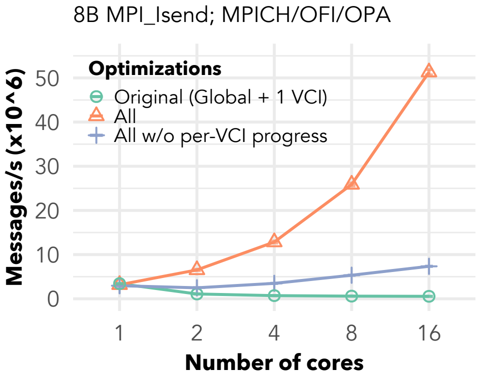


Figure 4.7: Effects of progress optimizations

initiates the reception of MPI_Ssend(comm1) by polling the single VCI that both communicators map to, thus allowing MPI_Ssend(comm1) to return. With multiple VCIs and per-VCI progress, MPI_Wait(req2) progresses only the VCI associated with comm2, preventing MPI_Ssend(comm1) to complete and causing deadlock. Figure 4.6 also describes a similar scenario with RMA operations using passive-target synchronization for cases where the underlying network requires target-side CPU involvement for progress.

In summary, the pure per-VCI progress model can improve performance, but the global progress model is necessary to ensure correctness even though it loses some performance.

To account for such communication patterns, we use a hybrid progress model; that is, we perform one round of global progress after a certain number of unsuccessful per-VCI progress attempts to complete an operation. We demonstrate the benefit of our hybrid per-VCI optimization in Figure 4.7. Communication throughput is $6.97\times$ 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. So, even when operations from multiple threads map to different VCIs, they contend on the request-class's lock when they need to acquire a request (e.g., during an `MPI_Isend`) or release it (e.g., during an `MPI_Wait`). To address this contention, we maintain a cache of requests for each VCI. Access to each cache is protected by the VCI's lock. During the creation of a request, we first attempt to acquire a request from the cache belonging to the VCI that the operation maps to. This does not require acquiring an extra lock because the lock for the VCI is already held for the operation. If the cache is empty, we fall back on acquiring a request from the global pool, which requires acquiring the request class's lock. The caching idea extends to releasing a request to the cache of a VCI as well. Thus, in the common case, we reduce the number of lock acquisitions in initiation operations to 1 (FG+per-VCI req-cache in Table 4.1, which summarizes the locks taken in different critical sections). Although the request class's lock is not taken (in the common case) for progress functions either, the VCI's lock pertaining to the request is taken twice—the final freeing of the request occurs in the MPI runtime layer, outside the critical section that protects the progress of the VCI.

In addition to traditional requests, MPICH maintains a pre-completed lightweight request for small-message transmissions. Up to a certain message size, modern intercon-

Table 4.1: Summary of locks on the critical path of initiation and progress operations in different critical sections.

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

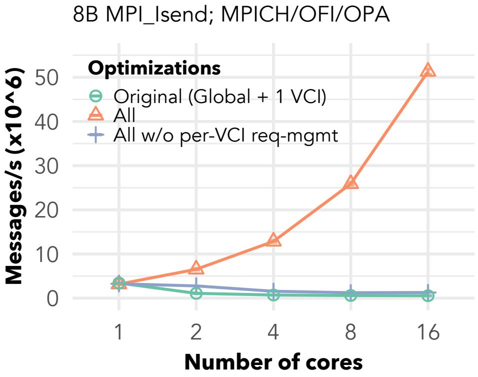


Figure 4.8: Effects of request-management optimizations

nects guarantee completion as soon as they are posted; they do not require any polling of the network.[5] MPICH optimizes memory usage for such operations by maintaining a global lightweight request that is marked as complete. These operations then simply increase the reference counter of the pre-completed request. A lightweight request is a single object and not a pool, so it cannot be cached like traditional requests. What we do instead is replicate this lightweight request and provide each VCI with its own. The per-VCI lightweight requests do not need atomic operations for their updates since each is protected by the lock of the VCI it belongs to.

Figure 4.8 shows the benefits of the per-VCI request management optimizations. Without the optimizations, throughput is $39.98\times$ lower (All w/o per-VCI req-mgmt) compared with all optimizations.

[5] A correct MPI implementation would need to poll the network intermittently even for such operations to progress any active message execution of an operation.

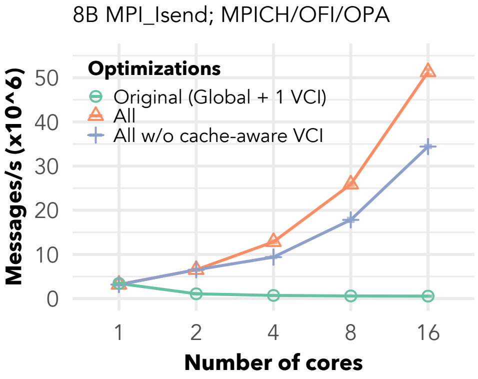


Figure 4.9: Effects of cache line awareness.

Cache-line Awareness

We implement the VCI pool as an array of structs. Each VCI struct holds the lock for that VCI. Locks of consecutive VCIs are likely to lie on the same cache line, resulting in the effects of false sharing when threads map to different VCIs. Hence, we use compiler attributes to cache-align each VCI. Figure 4.9 shows that without a cache-aware VCI, the message rate is $1.49\times$ lower (All w/o cache-aware VCI).

Summary

All the thread-safety and multi-VCI optimizations described in this section are critical for enabling fast parallel streams of communication for MPI+threads. The message rate achieved by the optimized MPI library with 16 threads for 8-byte `MPI_Isend`s is $94.43\times$ higher than that achieved by the original version of the MPI library.

4.5 Microbenchmark Analysis

We showcase the performance of the fast MPI+threads library on communication-intensive microbenchmarks for both point-to-point (`MPI_Isend`) and RMA (`MPI_Put`) operations. The benchmark demonstrates the maximum rate at which multiple cores can inject messages into the network simultaneously. Each core on the host node targets a distinct core on the remote node. We compare the following modes of execution.

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

For our analysis with MPI+threads, we spawn one rank per node with an OpenMP thread per core. MPI everywhere uses a rank per core. When users do not expose parallelism (ser_comm), all threads use the same communicator or window. In user-exposed parallelism (par_comm), each thread pair uses its own communicator or window.

4.5.1 Point-to-point communication

For the different modes of execution on OFI/OPA and UCX/IB, Figure 4.10 shows the message-rate scalability of a small-message `MPI_Isend`, and Figure 4.11 shows the message rate of `MPI_Isend` with 16 cores across varying message sizes. MPI everywhere achieves the highest throughput in all cases. When users expose communication parallelism, they achieve scaling communication throughput. When users expose no communication parallelism (ser_comm), however, there is no performance gain with increasing number of threads regardless of the optimizations in the MPI library.

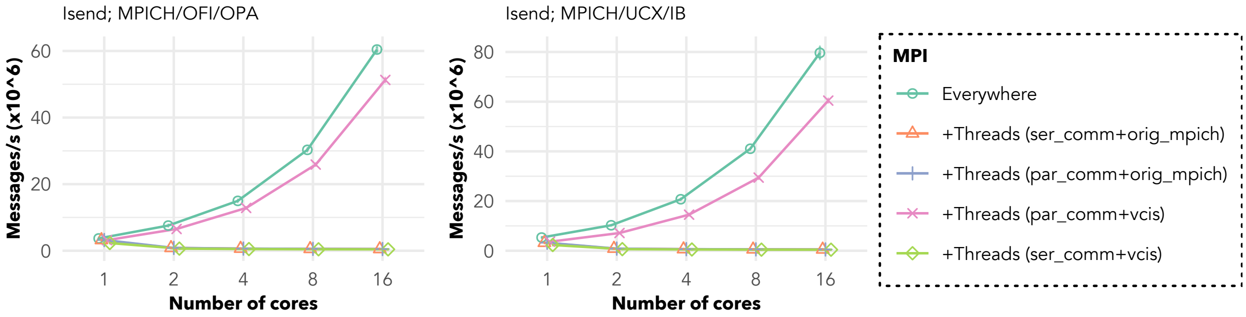


Figure 4.10: Message-rate scalability of 8-byte `MPI_Isend`.

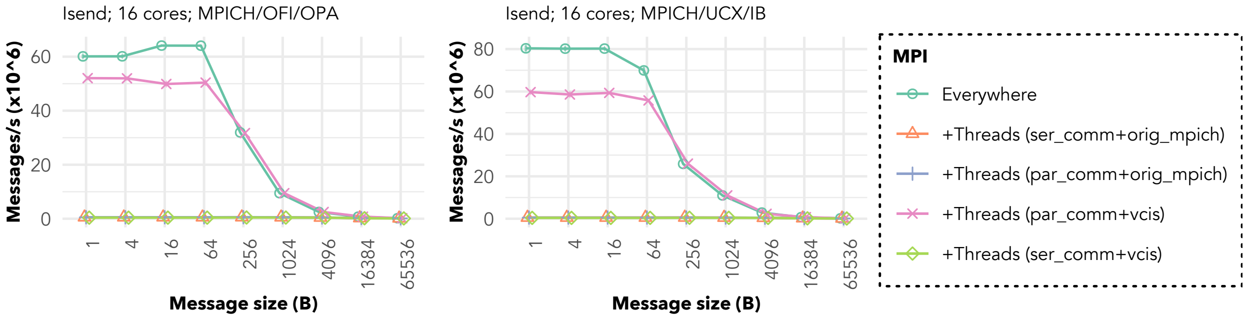


Figure 4.11: `MPI_Isend` throughput with varying message sizes.

4.5.2 RMA communication

Similar to Figures 4.10 and 4.11, Figures 4.12 and 4.13 demonstrate, for the different modes of execution on OFI/OPA and UCX/IB, the throughput scalability of a small-message `MPI_Put`, and the 16-core message rate of `MPI_Put` across varying message sizes, respectively.

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

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

The main point demonstrated here is the tradeoff between dedicated progress and shared progress. MPI everywhere has no distinction between dedicated and shared progress because it only has a single VCI. For MPI+threads, when a single VCI is used (*i.e.*, orig-

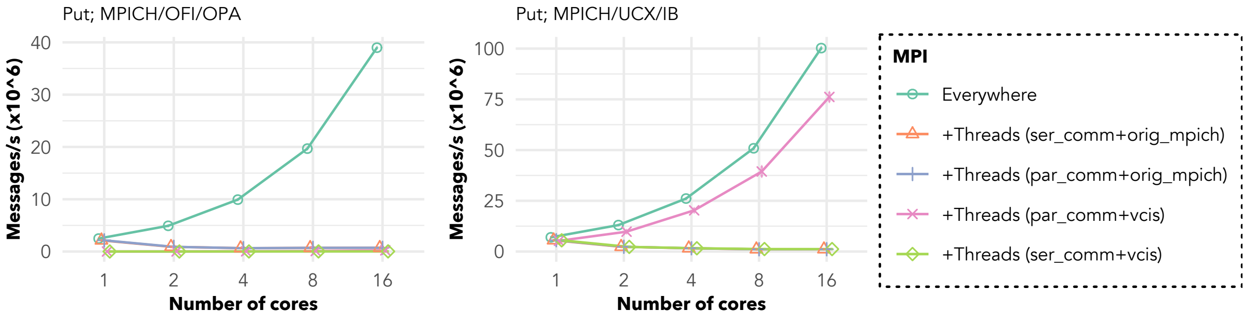


Figure 4.12: Message-rate scalability of 8-byte MPI_Put.

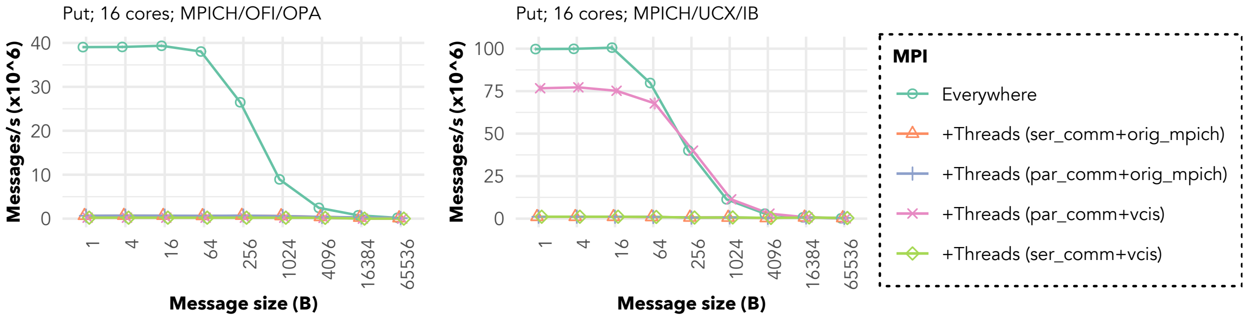


Figure 4.13: MPI_Put throughput with varying message sizes.

inal MPICH), like MPI everywhere, it has no distinction between dedicated and shared progress either. But, for MPI+threads, when we use multiple VCIs, the same independence of VCIs that enables good performance through the avoidance of locks also hurts shared progress between the threads. One can work around this issue by, for example, having each thread be responsible for progress on its window (in the same way that MPI everywhere works). One possibility is that threads call MPI_Win_free on their own windows in parallel (see Figure 4.14), thus making progress on the corresponding VCIs, although how practical this possibility is in real applications remains to be seen.

Busy target. Typically, the target side is involved in its own computational activities and does not just wait for communication to complete, as in Figure 4.14. The target's computation then determines the productivity of operations that need the target VCI to be progressed. Figure 4.15 shows a deteriorating MPI_Put message rate when the computa-

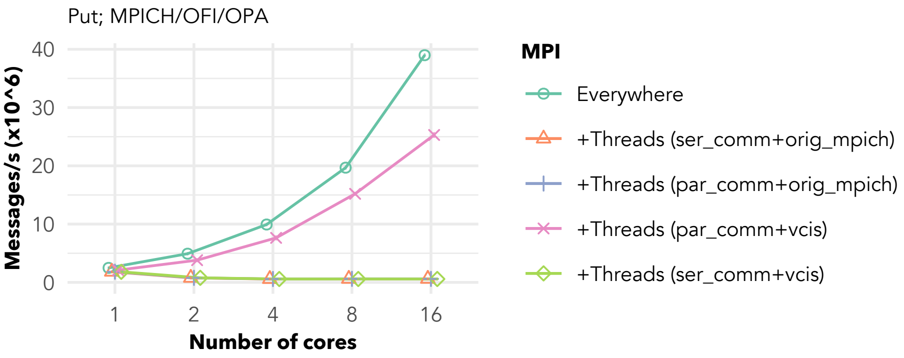


Figure 4.14: Parallel Win_free.

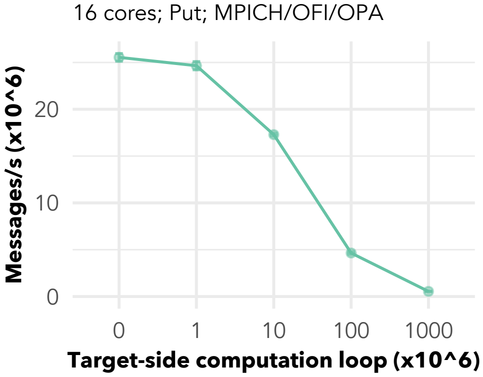


Figure 4.15: Busy target.

tion before the call to `MPI_Win_free` increases on the threads of the target rank.

4.5.3 Thread-safety costs

A corresponding MPI everywhere configuration represents the practical upper bound of the communication performance of an MPI+threads configuration. Our optimized MPI+threads library utilizes the same level of network parallelism as MPI everywhere. However, MPI+threads incurs thread safety overheads over MPI everywhere even in the uncontended case. These overheads are most visible for small messages (see Figure 4.11) since the message rate is bound by the CPU, not by the network. The sources of the thread

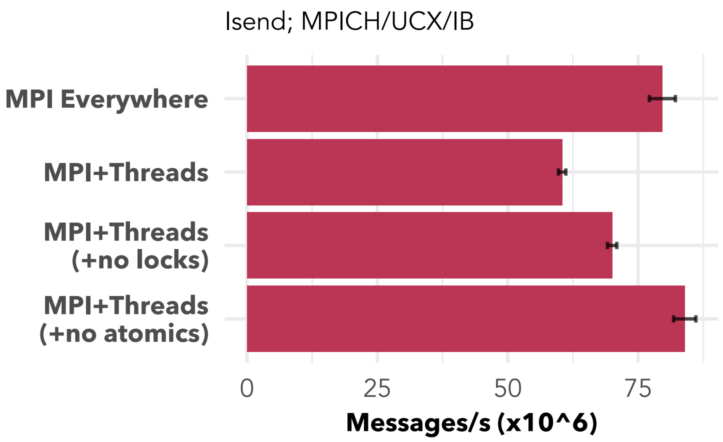


Figure 4.16: MPI+threads costs (note: error bars overlap).

safety overheads are lock acquisitions and atomics for completion or reference counting. Figure 4.16 shows that if we disable locking and atomics,[6] MPI+threads can match the throughput of MPI everywhere.

4.6 Communication Pattern Analysis with Mini-Apps

The effectiveness of using multiple VCIs in MPI+threads depends on the communication pattern of applications. In this regard, we classify MPI+threads communication patterns into three categories: (1) patterns that can directly use dedicated communication channels where VCIs saturate the network performance similarly to MPI everywhere; (2) patterns that require shared progress (see Section 4.5.2) where VCIs can suffer from loss in performance compared to MPI everywhere; and (3) patterns that need direct access to the network resources where using VCIs transparently through MPI mechanisms can hurt performance compared to MPI everywhere. Through mini-apps of applications, we study the communication performance of the three categories.

[6] Since each thread maps to its own VCI in the MPI+threads microbenchmark, disabling thread safety, although incorrect, does not lead to erroneous behavior.

4.6.1 Halo Communication in Stencil Applications

Stencils are arguably the most common design patterns in scientific computing applications. They are at the heart of various application domains such as computational fluid dynamics, image processing, and partial differential equation solvers. Prominent applications with the stencil communication pattern include Nek5000 [79] and LAMMPS [80].

Using a 2D 5-point stencil, we evaluate the neighborhood halo exchange (non-blocking point-to-point) time per iteration of the stencil pattern. We first partition the mesh into blocks across nodes, and then within each node we further partition the sub-block among cores (Figure 4.17 shows an example). The squares formed by the intersection of the dashed blue lines represent cores that are driven by processes in MPI everywhere and threads in MPI+threads parallelism. The blue dashed lines also represent boundaries where the halo exchange takes place through shared memory. MPI still executes intranode halo exchanges in MPI everywhere. In MPI+threads, threads use MPI only for internode halo exchanges and directly read the shared memory for intranode communication. The stencil pattern falls into the first category of applications—the internode communication of threads on edges of the nodes is independent and can execute on its own communication stream.

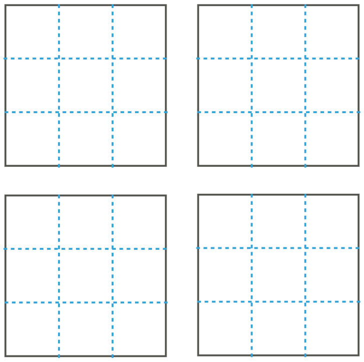


Figure 4.17: 6x6 grid with 3x3 sub-blocks per node.

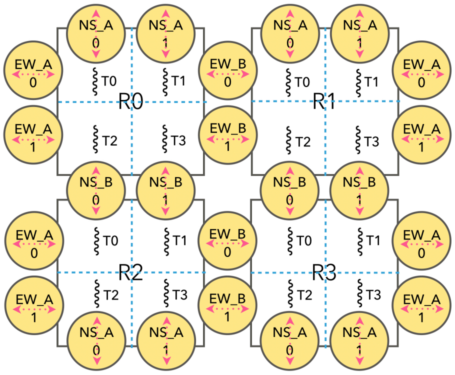


Figure 4.18: Logical parallelism with communicators in MPI+threads stencil.

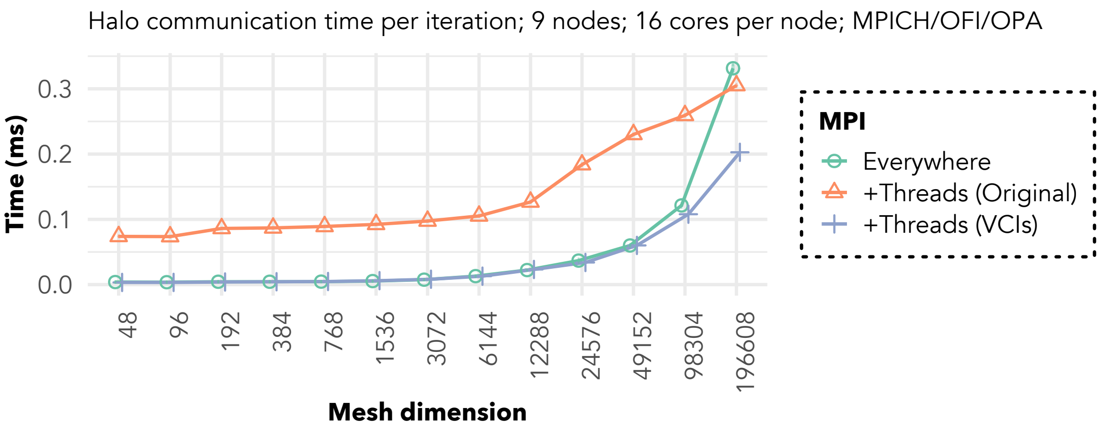


Figure 4.19: Halo communication across varying mesh sizes.

To expose the logically parallel point-to-point communication, we use two sets of communicators—odd and even—for each of the north-south and east-west exchanges. Each set contains as many communicators as there are threads on the node edge. Figure 4.18 shows an example. Depending on the Cartesian coordinates of the rank, the threads on a rank would use either the odd set or the even set. The odd-even sets prevent multiple threads from using the same communicator. Without them, T0 on R0 and R2 in Figure 4.18 would use the same NS_0 communicator, requiring T2 on R0 to also use the NS_0 communicator and thus serializing the communication of T0 and T2 on R0. Periodic stencils where the number of ranks along a dimension of the process-grid is odd require a separate set of communicators for the wraparound. The communicator usage can indeed be reduced without hurting performance by using only one communicator for the threads on corners, since their halo exchanges execute in serial.

Our evaluation utilizes all 9 nodes of the Skylake OFI/OPA cluster and engages 16 cores per node. Figure 4.19 shows the halo communication times for MPI everywhere and VCIs in MPI+threads across varying mesh dimensions. This time discards the cost of any load imbalance since we use MPI barriers before the start of each halo exchange. We observe that the communication performance of VCIs matches that of MPI everywhere parallelism.

4.6.2 Remote Data Fetches in OpenMC

The Center for Exascale Simulation of Advanced Reactors (CESAR) was a DOE co-design center whose primary objective was to adapt algorithms to the next-generation HPC architectures on the path to exascale systems. CESAR focused on algorithms that target the high-fidelity analysis of nuclear reactors. These include algorithms governing thermal hydraulics and neutronics. Applications simulating the former typically have a neighborhood, stencil style of communication, which we evaluated in Section 4.6.1. The latter consists of distributed Monte Carlo (MC) neutron-transport codes, such as OpenMC [89]. Siegel et al. [94] presented the original energy-banding (EB) algorithm for OpenMC, and Felker et al. [46] extended the EB idea to distributed-memory machines by distributing the cross-section data (composed of energy bands) across multiple nodes. Rather than the domain, particles are evenly distributed between the nodes. During simulation, each node fetches one band of the cross section using `MPI_Get` operations, tracks the movement of its share of particles, and iterates over the number of bands.

CESAR's EBMS miniapp [2] captures the communication pattern of the distributed EB idea. It utilizes MPI shared memory [56]: multiple processes on a node share a receive

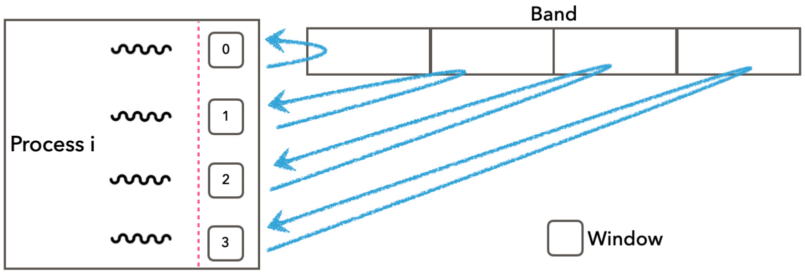


Figure 4.20: Communication parallelism in MPI+threads EBMS.

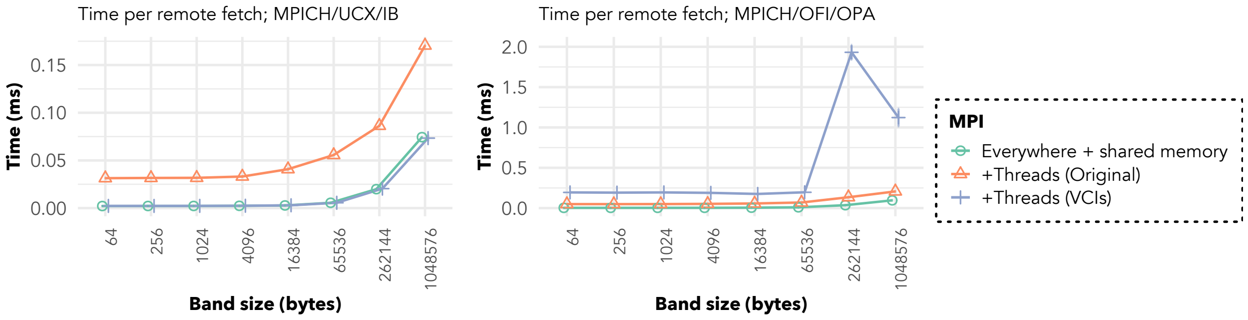


Figure 4.21: Time per remote fetch across varying band sizes with 16 cores per node on UCX/IB (left) and OFI/OPA (right).

buffer that is large enough to hold one band of the cross-section. While the computation is distributed among the different processes on the node, only one process is responsible for communication. We extended the EBMS miniapp to distribute the communication workload among the processes as well [3]. We also implemented a MPI+threads version of the miniapp with one multithreaded process per node. The communication workload between the cores is the same for both the MPI everywhere (+ shared memory) and the MPI+threads versions.

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

To leverage the independence between threads with MPI-3.1, we use a separate window per thread as shown in Figure 4.20. The windows point to the same memory, that is, the memory is not duplicated for each window.

Our evaluation utilizes 4 nodes and engages 16 cores per node on both the Gomez UCX/IB and Skylake OFI/OPA clusters. We measure the time for each fetch of a portion of

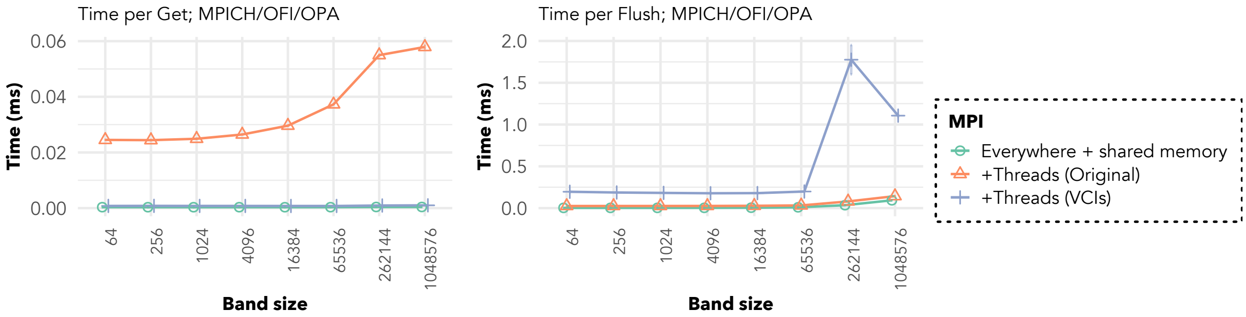


Figure 4.22: Get and flush time across varying band sizes on OFI/OPA.

a band that resides on a remote node. A remote fetch includes an `MPI_Get` and an `MPI_Win_flush`. Figure 4.21 shows the time for a remote fetch on the UCX/IB cluster. The communication performance of VCIs in MPI+threads is the same as that of MPI everywhere.

On the other hand, the remote-fetch times on Skylake OFI/OPA (see Figure 4.21) show that exposing parallelism in MPI+threads hurts performance, especially for large messages. The time for a remote fetch is governed by the issue of the fetch (`MPI_Get`) and its completion (`MPI_Win_flush`). If we separate them out, Figure 4.22 shows that the time for an `MPI_Get` using multiple VCIs is the same as that in MPI everywhere but the time of `MPI_Win_flush` is more expensive. The reason is that the communication pattern of the application does not guarantee that the remote VCI being targeted by the `MPI_Get` operations will be progressed—the thread mapped to the target VCI on the remote rank could be waiting on a thread-barrier that exists between each iteration of the simulation. Intel OPA relies on the application to make progress on the target VCI for the completion of large-message RMA transfers and for a productive execution of small to medium message transfers. Hence, the execution is dependent on the occasional global progress in the progress engine (see *Per-VCI Progress* in Section 4.4.1).

4.6.3 Block-Sparse Matrix Multiplication in NWChem

NWChem [101] is a prominent quantum chemistry application suite for large-scale simulations of chemical and biological systems. To distribute the multidimensional arrays across the memories of multiple nodes, it uses the Global Arrays (GA) [75] library, which provides access to remote data through one-sided MPI operations. When NWChem is used for quantum chemical many-body methods, such as CCSD and CCSD(T), the dominant cost is that of BSPMM: block-sparse matrix multiplication (tensor contractions). NWChem implements BSPMM with dense matrix operations using a *get-compute-update* pattern: each worker (processing entity) uses `MPI_Get` to retrieve the submatrices it needs, and after the multiplication it uses an `MPI_Accumulate` to update the memory at the target location.

Using a mini-app [1], we evaluate a 2D version of this communication pattern that performs $A \times B = C$, wherein the input matrices A and B are composed of tiles. Each tile is either a dense or zero matrix. The nonzero tiles are evenly distributed among the ranks in a round-robin fashion. Each rank maintains a work-unit table that lists all the multiplication operations that workers need in order to cooperatively execute. Rank 0 hosts a global counter, which the workers fetch and add atomically (`MPI_Fetch_and_op`). The fetched counter serves as an index to the work-unit table. Each worker locally accumulates its C tiles until the next fetched work unit corresponds to a different C tile, in which case the worker uses an `MPI_Accumulate` to update the C tile. A worker is a process in MPI

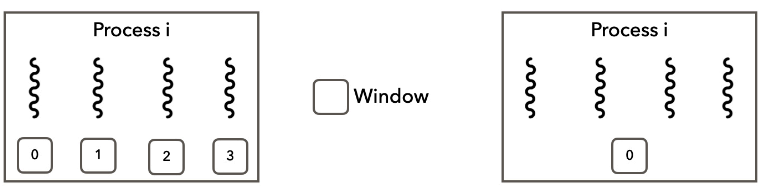


Figure 4.23: Logical parallelism in MPI+threads BSPMM.

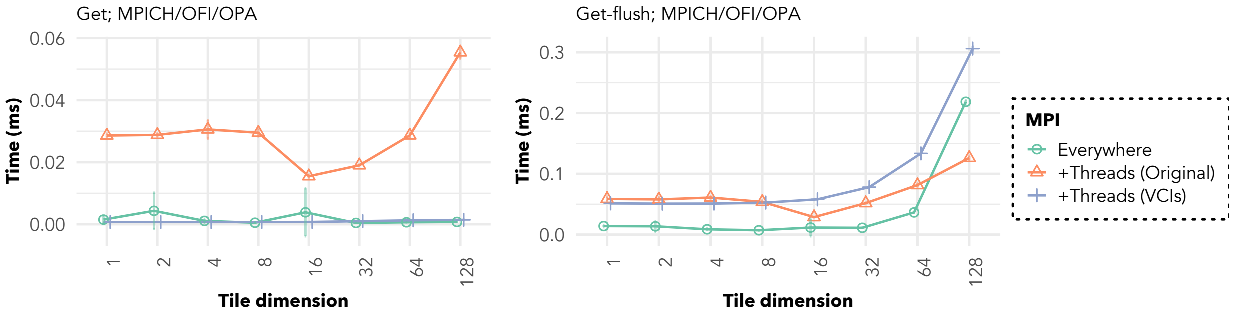


Figure 4.24: BSPMM Get communication performance on Intel Omni-Path.

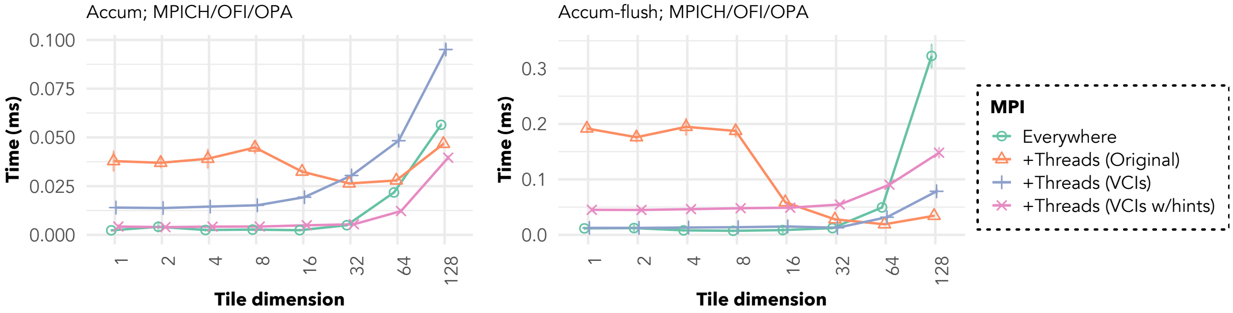


Figure 4.25: BSPMM Accumulate communication performance on Intel Omni-Path.

everywhere and a thread in MPI+threads.

MPI+threads BSPMM's communication pattern falls under the third category. Although each thread can use its own window for its `MPI_Get` to fetch tiles of A and B, MPI-3.1's default semantics constrain the threads within a process to use a single window for the `MPI_Accumulate`. Each thread cannot use its own window for `MPI_Accumulate` because atomicity across windows for the same memory location is undefined.

Figure 4.24 and Figure 4.25 portray the performance of BSPMM's communication pattern on 4 nodes of the Skylake OFI/OPA cluster with 16 cores engaged per node. We measure the time taken to initiate the operations (e.g., `MPI_Get`) separately from the time taken to complete them (e.g., `MPI_Win_flush`). VCIs initiate `MPI_Get` operations as fast as MPI everywhere. However, for `MPI_Accumulate` operations, MPI+threads with

MPI-3.1 is constrained by the use of a single window, so VCIs issue operations slower than MPI everywhere. The flush of MPI_Get operations demonstrates behavior similar to that in the EBMS pattern (see Section 4.6.2). MPI+threads with a single window flushes MPI_Accumulate operations as fast as MPI everywhere for small and medium tiles because of the use of a single VCI—the probability of the remote target VCI being progressed is higher since all threads on the target rank map to it. The Accum-flush of MPI everywhere, however, is slow for large tile dimensions because the worker (process) cannot progress its VCI until it finishes its computational tasks, which is larger for large tile dimensions. On the other hand, if a worker in MPI+threads is busy with computational tasks, other workers (threads) on the same process might progress the VCI, either because they all map to the same VCI or because of shared progress between threads, allowing for a more productive execution of a large-message RMA operation than that in MPI everywhere.

An important point to note is that the MPI_Accumulate operations in BSPMM do not need to be ordered. Hence, if the user hints this relaxation using the accumulate_ordering=none hint, the MPI library can issue the operations from different threads in parallel using automatic mapping to VCIs and thereby achieve similar performance as MPI everywhere in issuing operations, as we can see in Figure 4.25. For Accum-flush, however, VCIs with hints suffer compared to a single VCI because the probability of the remote VCIs being progressed is lesser when multiple VCIs exist. Our previous work did not capitalize on this hint [107].

4.7 Related Work

The communication performance of MPI+threads has been a decade-long concern. Researchers have studied the problem in various ways, ranging from mitigating lock contention on the MPI library's software resources [27, 26, 32] to extending the MPI standard [43, 51]. We discuss prior works that are conceptually related to ours.

4.7.1 MPI Endpoints demonstration

Dinan et al. [44] and Sridharan et al. [95] demonstrate scaling performance of MPI+threads with the MPI Endpoints proposal (we discuss this mechanism of exposing communication parallelism in further detail in Section 5.2.1). Their works, however, do not describe the designs of changes required to the MPI library to enable scalable MPI+threads communication. Additionally, their work does not describe the notion of shared progress between threads, which is critical for correctness. Our work, on the other hand, does not sacrifice correctness for performance.

4.7.2 Intel MPI

Since its 2019 release, the Intel MPI library has utilized multiple network hardware contexts on Intel Omni-Path through its multiple endpoints support [8]. However, this support is only for a nonstandard threading level: MPI_THREAD_SPLIT, which does not

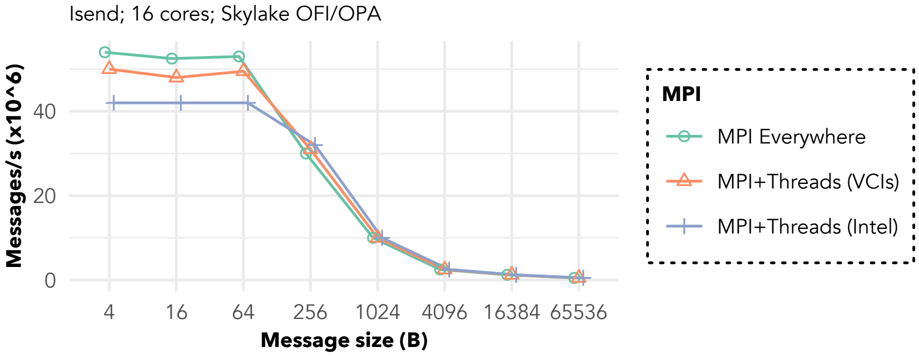


Figure 4.26: VCIs compared against Intel MPI.

cover all cases possible in the MPI_THREAD_MULTIPLE threading level. In contrast, our work with VCIs supports MPI_THREAD_MULTIPLE fully and correctly. Nevertheless, VCIs outperform the multiple-endpoints support in Intel MPI, as we can see in Figure 4.26

4.7.3 Open MPI

A couple of works [50, 78] on Open MPI are conceptually similar to our work—they use fine-grained critical sections and map parallelism available in the existing MPI standard to multiple network hardware contexts to improve MPI+threads communication performance. However, both works do not compare against MPI everywhere. Additionally, like the MPI Endpoints work, neither of these works discusses the notion of shared progress, ignoring correctness.

Gopalkrishnan et al. [50] evaluate the communication performance of MPI+threads with OFI scalable endpoints. Recognizing the practical performance limitations of scalable endpoints, our work uses regular OFI endpoints instead, and hence we observe much larger speedups than their work obtains.

Similar to VCIs in our work, Patinyasakdikul et al. [78] define Communication Resource Instances (CRIs). Their approach involves creating a pool of CRIs and either assigning CRIs to operations in a round-robin fashion or assigning CRIs to threads using thread-local storage. While this approach may be correct for a subset of operations, some CRIs break MPI's semantics for operations such as `MPI_Accumulate` operations to the same target location. Such operations are ordered by default on a window. In terms of performance, even with user-exposed parallelism their point-to-point communication performance does not scale with increasing number of threads unlike the results of our work.

4.8 Concluding Remarks

For both point-to-point and RMA operations, an MPI library is able to achieve scaling communication throughput by establishing fast parallel communication streams. These parallel communication streams are a result of utilizing the network parallelism available on the NICs of modern interconnects and enabling fast multithreaded access to the MPI library's software resources. MPI+threads applications can utilize such fast parallel communication streams by exposing the logical parallelism in their multithreaded communication to the MPI library. In the next chapter, we discuss the various venues in which domain scientists can express logically parallel communication.

Chapter 5

Unlocking MPI+Threads Applications with Logically Parallel Communication

In this chapter, we utilize the fast MPI+threads library that we developed in Chapter 4 to unlock the true potential of the MPI+threads programming model for applications. The key contributing factor of a fast MPI+threads library is the proactive use of network parallelism inside the library. By mapping to multiple network hardware contexts per process, MPI libraries can funnel parallel independent communication through distinct network contexts. The MPI libraries, however, are helpless if domain scientists do not distinguish between operations that are ordered and those that are independent. The MPI standard enforces certain ordering constraints that the MPI library must adhere to for correctness. Hence, if the MPI communication from multiple threads is independent, application developers must identify this independence in its MPI communication and expose *logical communication parallelism* to the MPI library. Only the application can provide such information to the MPI library. How then can domain scientists utilize network parallelism in their MPI+threads applications? The MPI community holds two schools of thought in this regard.

The earlier school of thought believes in providing domain scientists with user-visible endpoints to express parallelism by creating multiple ranks per process [95, 44, 58]. While endpoints are a flexible mechanism for applications to control their multithreaded communication, the challenge with introducing endpoints is intrusive changes to the MPI standard in the form of new APIs [43, 11]. The MPI forum has deliberated the MPI Endpoints proposal but ultimately suspended the consideration of the proposal since the opposing school of thought noted that the existing MPI semantics already allow domain scientists to expose logically parallel communication. So, why not capitalize on the capabilities of existing mechanisms? The recent school of thought advocates for the use of existing MPI mechanisms such as communicators, tags, and windows. The challenge with this approach is that existing MPI semantics can sometimes prevent users from exposing the logical parallelism available in an application even if the app does not rely on the limiting semantic. To work around this issue, the MPI forum has voted in new MPI Info hints in the draft MPI-4.0 standard that allow domain scientists to inform the MPI library which semantics an application does not need. Not only does such information allow the MPI library to optimize communication, but it also provides domain scientists with more opportunities to express parallelism with existing MPI mechanisms. Compared to the API changes required for user-visible endpoints, the introduction of hints are less intrusive extensions to the standard.

In this chapter, we present an overview of the mechanisms that domain scientists can use to expose logically parallel communication both with the existing MPI-3.1 standard and the potential future iterations of the standard (Section 5.2). We study the performance implications of the different mechanisms on the end-to-end runtime of applications (Section 5.3).

5.1 Evaluation Testbeds

For our evaluation with applications, we use our MPI implementation (based on the newest MPICH) from Chapter 4 that supports high-speed multithreaded communication.

Our testbeds include two small-scale platforms and two medium-to-large-scale platforms. The small-scale platforms are the same as those used in Section 4.2. The Skylake testbed consists of nodes with Intel Skylake nodes that are interconnected with Intel Omni-Path (OPA), and the Gomez testbed consists of Intel Haswell nodes that are interconnected with Mellanox InfiniBand EDR (IB). The medium-to-large-scale platforms include the Bebop HPC system at Argonne National Laboratory and the HPC3 cluster at UC Irvine. Bebop features two clusters: one with Intel Knights Landing (KNL) CPUs and another with Intel Broadwell processors. Both clusters use the Intel OPA interconnect. HPC3 features Mellanox IB. For Skylake and Bebop, we use MPCIH/CH4's OFI `netmod` in conjuction with PSM2 (OFI/OPA); for Gomez and HPC3, we use the UCX `netmod` with Verbs (UCX/IB).

5.2 Mechanisms to Expose Logical Parallelism

5.2.1 User-Visible Endpoints

The MPI Endpoints proposal to extend the MPI standard allows the user to create multiple MPI ranks per process by creating new communicators and windows with multiple endpoints [11]. As is the case with messages from different MPI processes, messages using different endpoints in a process have no order between them. Hence, by using different endpoints for an application's logically parallel communication, domain scientists would expose logical communication parallelism within a process to the MPI library.

Since endpoints take on the semantics of an MPI rank, they are directly addressable, giving users flexible control over the communication between endpoints. In other words, if the user maps each thread to a distinct endpoint, then all threads are directly addressable.

The MPI library would then, in theory, map each endpoint to a dedicated communication channel (a VCI from Chapter 4). This way, applications can establish dedicated communication channels to the network for each thread with a distinct endpoint per thread. It is important to note 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, depending on the availability of network resources, could be fewer than the number of endpoints.

The endpoints interface, albeit flexible, requires intrusive extensions to the standard in the form of new APIs. `MPI_Comm_create_endpoints` in the draft of the proposal [11] allows the user to create a new communicator with a user-specified number of endpoints. It returns an array of handles to a new communicator, each directly addressable through a distinct rank in the new communicator. If the point-to-point operations of threads are logically parallel, the user would use a distinct communicator-handle (endpoint) in its point-to-point operation (e.g., `MPI_Isend`) and specify the rank of remote endpoint that it intends to target. The draft of the proposal contains a detailed explanation of the API [11]. Although this draft does not describe an API to create an RMA window with multiple endpoints, the idea would be similar to that of `MPI_Comm_create_endpoints` (e.g., `MPI_Win_create_endpoints`) where the user would receive an array of handles to a new window.

5.2.2 Existing MPI Mechanisms

For both two- and one-sided MPI communication, existing MPI semantics allow applications to expose communication parallelism, as we saw in Section 4.1. MPI-3.1 provides users with many opportunities to express parallelism for RMA operations (through distinct windows, or through multithreaded communication of unordered operations within a window). But for point-to-point operations, MPI-3.1 allows the expression of logically parallel communication for point-to-point operations only through communicators, which can be limiting for applications especially when the application does not rely on the MPI semantics (non-overtaking order, or wildcards) that prevents the use of other existing MPI mechanisms such as ranks and tags.

The Draft MPI-4.0 Standard

To overcome the limitations for expressing point-to-point communication parallelism in the next iteration of the standard, MPI-4.0, the MPI forum has voted in new MPI Info hints that will allow applications to relax the MPI semantics it does not need. Not only does such relaxation of semantics yield new opportunities for performance optimizations in the MPI library, but it also opens up new opportunities for the application to express logically parallel communication.

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` [10]. 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 in (relaxing the non-overtaking order). These hints mean that users can express logically parallel communication within a communicator by using distinct ranks and tags for each thread. Note that the matching semantics of MPI still apply with these hints, that is, the ⟨communicator, rank, tag⟩ envelope of messages must still match for successful communication.

5.2.3 Mapping Mechanisms to Network Parallelism

In Section 4.4 of Chapter 4, we learned that a fast MPI+threads library maps logically parallel communication. Here, we expand upon how our implementation of the MPI library maps the mechanisms of expressing parallelism to its VCIs.

In the endpoints mechanism, we map the communication from distinct endpoints to distinct VCIs. This mapping gives the user 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 (further discussed in Chapter 6), the possibility of a mismatch in expected mapping to VCIs is more likely with existing MPI mechanisms than with user-visible endpoints.

To allow users to better control the mapping of their logically parallel communication to VCIs with existing MPI mechanisms, we introduce a new set of MPI Info hints complementing those introduced in the MPI-4.0 draft standard. The new set of hints does not influence any MPI semantic, rather it is specific to our MPI implementation.[1] 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. As discussed in Section 4.4, by default, if the user provides no MPI Info hints, all communicators and windows map to the same single VCI. By using Info hints, however, applications can request a new VCI or multiple new VCIs for a communicator or window.

[1] Introducing new implementation-specific Info hints does not violate the MPI-3.1 standard.

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 the user to inform the MPI library which bits of the tag they will use to express logical parallelism. 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 for non-atomic operations with existing MPI mechanisms, users have the option to either let the MPI library automatically exploit their multithreaded RMA operations or explicitly express the communication parallelism using separate windows. For such operations, we allow the user to choose whether they prefer to use automatic mapping or explicit mapping to VCIs through an Info hint. For atomic RMA operations, users only have the option of exposing parallelism through multiple windows unless they hint to the MPI library that the order of atomic operations on a window does not matter, in which case they could use automatic mapping. The nature of automatic mapping, however, incurs the overheads of hash collisions.When applications use explicit mapping, all the operations of a window funnel through the VCI mapped to that window.

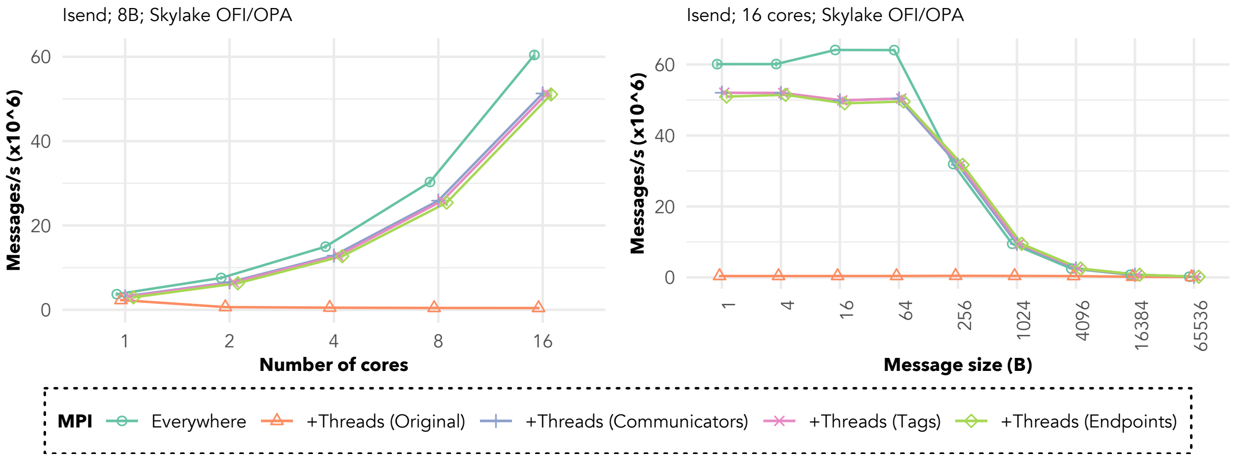


Figure 5.1: Microbenchmark performance comparison of mechanisms to expose logically parallel point-to-point operations.

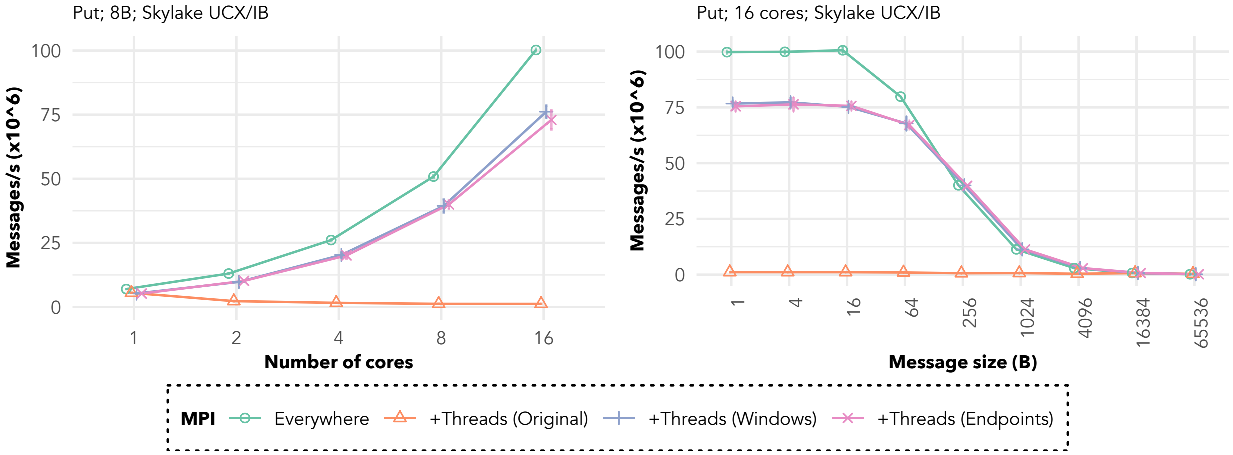


Figure 5.2: Microbenchmark performance comparison of mechanisms to expose logically parallel RMA operations.

5.2.4 Microbenchmark evaluation

Figure 5.1 and Figure 5.2 compare the performance differences of the different mechanisms to expose logically parallel communication for point-to-point and RMA operations, respectively, on communication-intensive microbenchmarks. Given their flexible interface, user-visible endpoints represent the upper bound in terms of exposing logical communication parallelism for a given communication pattern. For the simple communication pattern of the message-rate benchmark, existing MPI mechanisms are able to expose all of the available communication parallelism like user-visible endpoints. Hence, all mechanisms perform equally well as we can see from the figures. When the communication performance is bounded by the injection overhead of the CPU (as is the case in small-message communication), all MPI+threads mechanisms perform slower than the corresponding MPI everywhere parallelism because thread-safety costs add to the injection overhead (see Section 4.5.3).

5.3 Application Case Studies

In this section, we demonstrate the impact of dissolving the communication bottleneck on the end-to-end runtime of applications. We pick applications and computational frameworks from three different domains—computational fluid dynamics (Uintah), astrophysics (WOMBAT), and data-centric programming systems (Legion)—that have been designed to scale on the upcoming exascale machines.

5.3.1 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 [34]. 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.

Challenges

Two components play key roles in ARCHES' boiler simulations: Reverse Monte Carlo Ray Tracing (RMCRT) [60] that solves the radiation transport equation, and the hypre library [45] that solves a large system of linear equations projecting pressure at every timestep. RMCRT has been extensively developed to scale [73, 59, 57], 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 rays through the entire domain. Given the data duplication per node with MPI everywhere, RMCRT runs out of memory for large domains. For example, the weak-scaling study with 1 patch per core in Figure 5.3 shows that Uintah runs out of memory for the domain size at 64 nodes with MPI everywhere. 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 [74]. 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

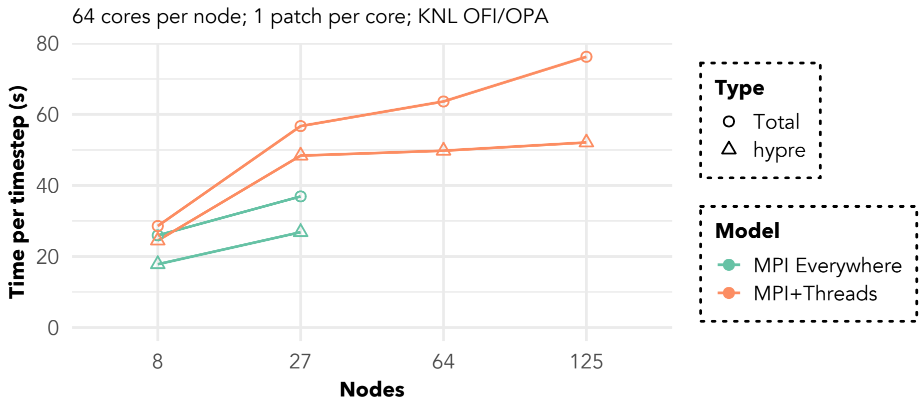


Figure 5.3: MPI everywhere vs. MPI+threads in Uintah.

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. [91] 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 logically parallel in this pattern, hypre still performs slower with MPI+threads than with MPI everywhere because of a higher MPI time (see Figure 5.4). hypre spends more time in MPI with MPI+threads because it does not expose its logically

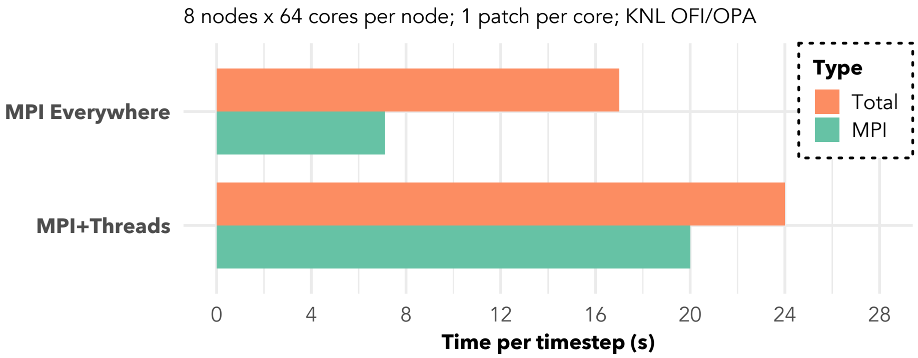


Figure 5.4: Higher MPI time in MPI+threads hypre.

parallel communication 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; described in the next subsection). Thus, even though hypre could expose 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. However, 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 theoretically independent MPI 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.

Unlocking Uintah and hypre with Logically Parallel Communication

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 3D patches between cores statically for both MPI+threads and MPI everywhere parallelism.

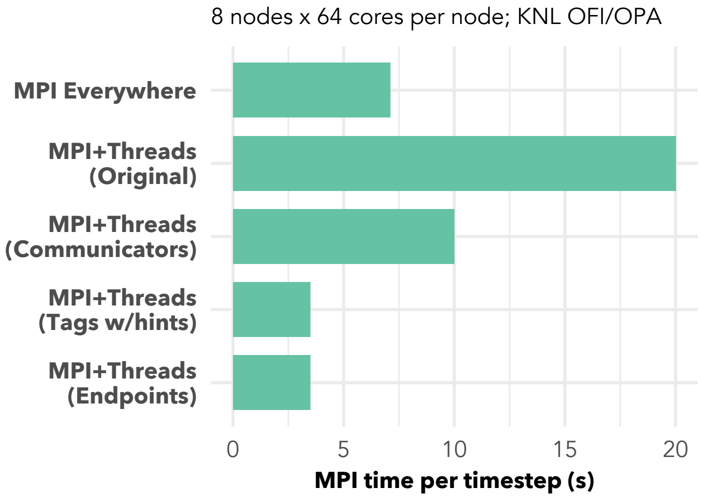
User-visible endpoints. We can express maximal communication independence with user-visible endpoints by creating as many endpoints as there are threads. Each thread uses its local endpoint to issue operations, and it communicates with the remote thread using the rank of the target endpoint. On a KNL 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. 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.

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 (described in the previous subsection). 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 5.2.3, 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.

Figure 5.5 compares the performance of the different mechanisms of expressing logically parallel communication 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



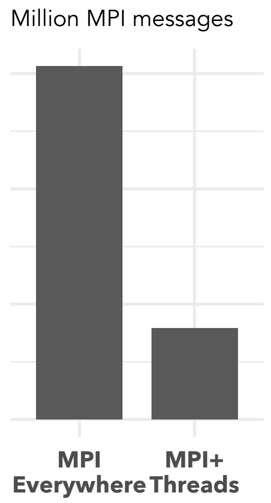


Figure 5.5: Different mechanisms of exposing logical communication parallelism for hypre (1 patch per core).

Figure 5.6: MPI communication volume per node.

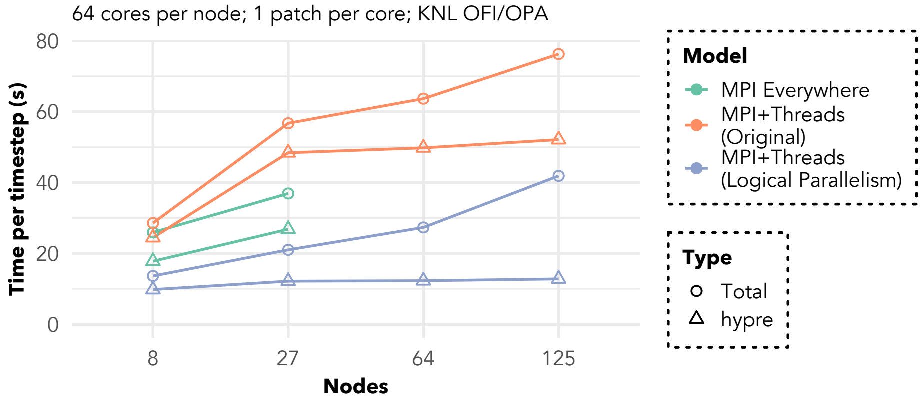


Figure 5.7: Uintah with logically parallel communication.

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 5.6.

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. The weak-scaling study with 1 patch per core in Figure 5.7 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. Figure 5.7 also shows that logically parallel communication improves the weak-scaling parallel efficiency of hypre, but not that of Uintah, indicating that the next bottleneck in improving Uintah's parallel performance is RMCRT.

5.3.2 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 [18]. 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 [72]. 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 environ-

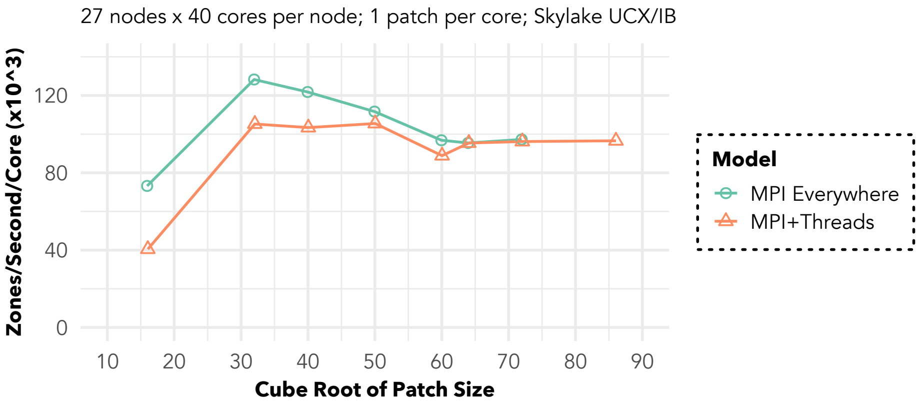


Figure 5.8: MPI everywhere vs. MPI+threads WOMBAT.

ment 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 5.8 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.

Challenges

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

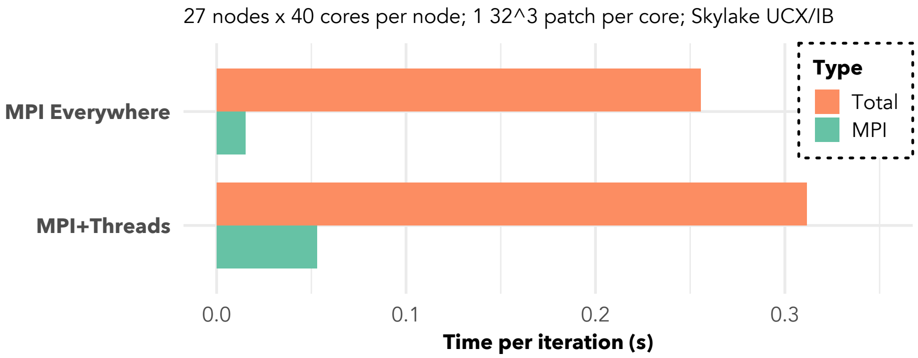
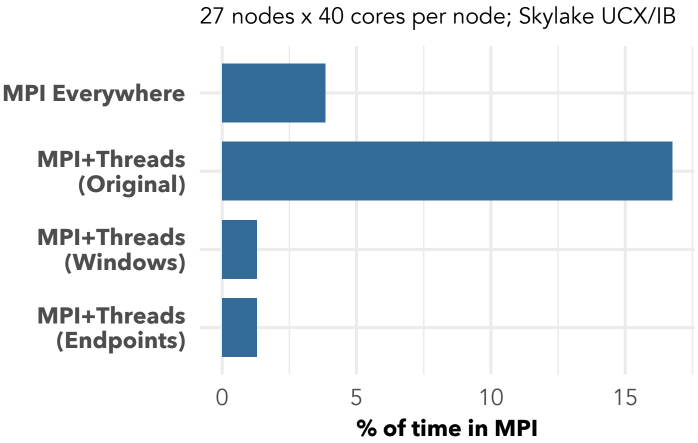


Figure 5.9: Higher MPI time in MPI+threads WOMBAT.

not explicitly express any of the available logical parallelism in its multithreaded MPI communication—all operations use a single window. Although the current MPI standard does not maintain ordering constraints for nonatomic RMA operations such as `MPI_Put` and `MPI_Get`, the parallel execution of such operations is at the mercy of the MPI library. Due to the conservative approaches in existing MPI libraries, WOMBAT spends more time in MPI with MPI+threads than with MPI everywhere (see Figure 5.9). Even if the MPI library optimizes for multithreaded RMA communication, without any communication parallelism information from the application the MPI library is forced to use some flavor of hashing to funnel communication from threads to the limited underlying network channels. As is the case with any hashing, collisions hurt performance. Hence, to eliminate the communication bottleneck in MPI+threads, WOMBAT needs to efficiently expose the logical parallelism in its multithreaded communication.

Unlocking WOMBAT with Logically Parallel Communication

Since the parallel operations of WOMBAT are logically independent, one can eliminate the communication bottleneck by explicitly exposing the logically parallel communication to a fast MPI+threads library.



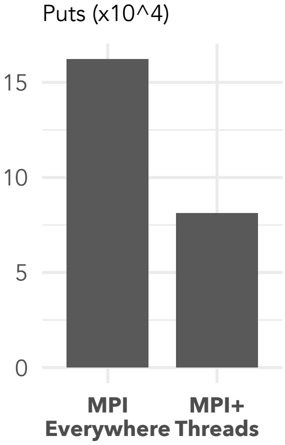


Figure 5.10: Different mechanisms of exposing logical communication parallelism for WOMBAT (1 32ˆ3 patch per core).

Figure 5.11: MPI Puts per node.

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 evaluated the publicly available version of WOMBAT [21] 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 5.10 shows that utilizing network 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 5.11 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 5.12 shows that the reduction in MPI time with logically parallel communication boosts the overall scientific throughput of MPI+threads, especially for smaller 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 [72].

5.3.3 Legion

In modern heterogeneous computing environments, the cost of data movement dictates the overall computational efficiency of an application. Hence, to achieve high perfor-

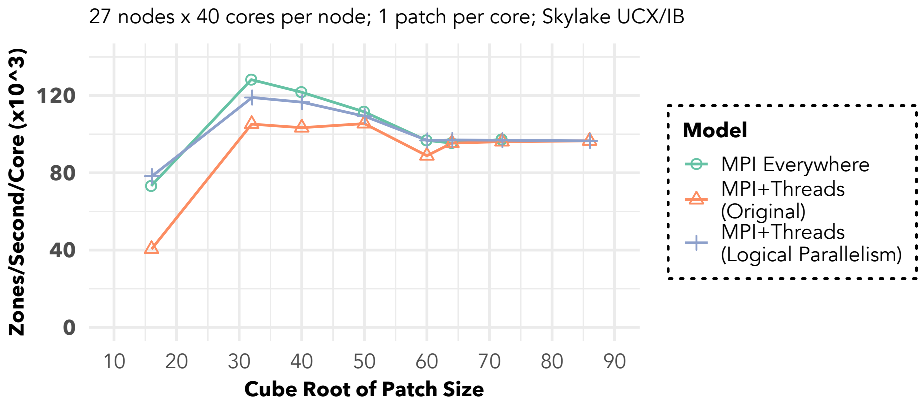


Figure 5.12: Scientific throughput of WOMBAT with logically parallel communication.

mance, 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 [33]. 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 [99]. 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.

Challenges

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 [105]. 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 researched 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 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 [33, 98].

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

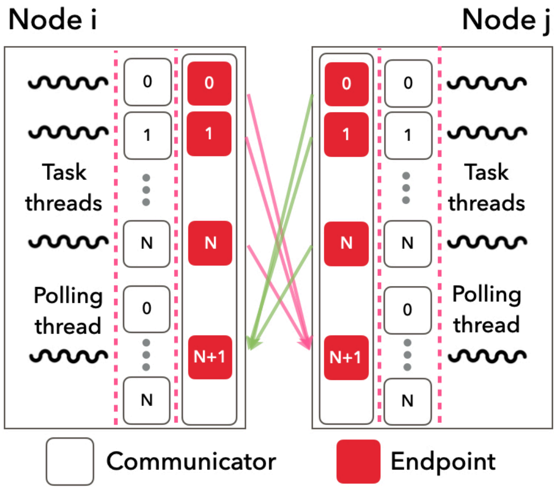


Figure 5.13: Limiting existing MPI semantics compared to user-visible endpoints for the communication pattern in Legion applications.

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.

Unlocking Legion with Logically Parallel Communication

Without exposed communication parallelism, the single 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 can see in Figure 5.13, 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 hints. With this information conveyed, we can expose the communication parallelism between task threads by encoding, for example, their thread IDs into 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 evaluated the different mechanisms of expressing communication parallelism in Le-

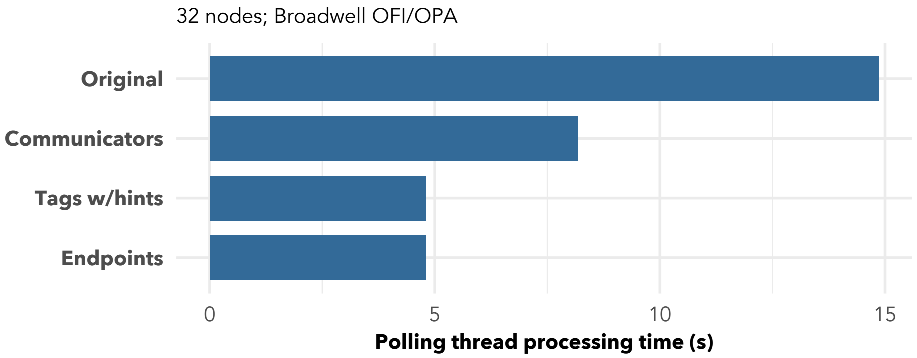


Figure 5.14: Polling thread processing on Circuit's critical path.

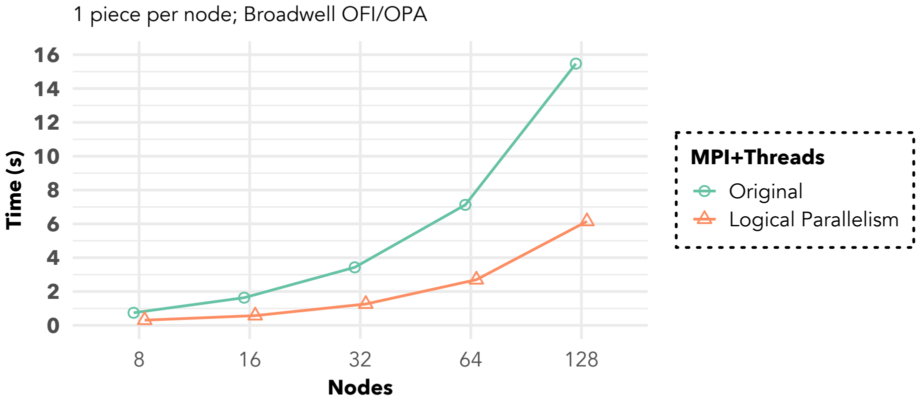


Figure 5.15: Circuit simulation performance with logically parallel communication.

gion's MPI backend for the Circuit simulation application [33] on Bebop's Broadwell OFI/OPA cluster. Figure 5.14 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 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.

Figure 5.15 shows the overall performance of the Circuit simulation with workloads at the strong-scaling limit (1 piece per node with 2 nodes per piece and 4 wires per piece). 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 5.15 shows that we can improve the Circuit simulation performance by 2.62× on average. We would like to note that the problem of poor weak-scaling efficiency in both versions is orthogonal to the problem of exposing the communication independence between the threads. This work addresses the latter problem.

5.4 Concluding Remarks

By exposing the communication independence between the threads of an application through *logically parallel communication*, we eliminate the multithreaded communication performance bottlenecks present in today's applications that have been designed to run on the modern many-core processors. We do so by leveraging the new opportunities of using existing MPI mechanisms that will soon (2021) be available in MPI-4.0, the next iteration of the MPI standard. With MPI-4.0, existing MPI mechanisms perform as well as the upper bound set by user-visible endpoints. Communication-intensive applications realize a significant boost (over 2×) in their overall performance when the MPI 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. Eliminating the contention between threads for communication operations shows that other algorithmic and threading challenges (e.g., mitigating thread-synchronization overheads) are the next critical bottlenecks in MPI+threads applications. We encourage domain experts to address these challenges in their MPI+threads codes to achieve both higher scalability and higher productivity over the traditional MPI everywhere model.

Chapter 6

If Not MPI Endpoints, How About MPI Rankpoints?

Chapter 5 shows that the benefits of exposing logically parallel communication in MPI+threads application goes beyond the utilization of the underlying network parallelism. By dissolving the multithreaded communication performance bottleneck, logically parallel communication helps applications maximize the benefits of the MPI+threads programming model over their traditional MPI everywhere versions. A key observation in our application studies is that existing MPI mechanisms expose the same level of logically parallel communication as does the flexible interface of user-visible endpoints. If the performances of both approaches are the same, which approach should the supercomputing community then pursue? Presumably, the solution that does not intrusively change the standard through new APIs would be the preferred choice.

However, during our collaborations with domain scientists for application case studies, we learned that expressing logical parallelism in some MPI+threads applications, in particular those that use point-to-point communication, through communicators is quite complex. The complexity hurts not just the productivity of the application developer but also the MPI library's mapping to network parallelism, thereby hurting performance. One way to combat this complexity is to relax MPI semantics through hints which would allow users to express parallelism through other mechanisms such as ranks and tags (the way the draft MPI-4.0 standard is doing). Such a solution, however, does not uniformly apply to all types of MPI communication (e.g., collectives). For the same application, expressing parallelism through user-visible endpoints is much more straightforward because of the flexibility of its design (each endpoint is directly addressable) and familiarity of domain scientists to program with MPI ranks (each endpoint takes on the semantics of a rank).

The productivity concern with user-visible endpoints is that it introduces a new concept through new APIs to the user. Presumably, this concern does not hold with MPI-3.1 since users are already aware of communicators and windows. However, our discussions with domain scientists demonstrate that MPI users do not intuitively associate existing MPI objects as a means of expressing parallelism. For example, users have historically viewed communicators as groups of processes. That it can double up as a means to express logical parallelism is a corollary of the definition of a communicator. Hence, the concern of enforcing new concepts on users holds for MPI-3.1 too.

Like with the success of any technology, the answer to the posed question lies within the end users, the domain scientists. In this chapter, we compare the strengths and limitations of the designs of each approach with respect to their applicability to MPI's point-to-point, RMA, and collective communication. We summarize the opinions, thoughts and concerns that we have gathered from our collaborations with several domain scientists who represent different types of MPI+threads applications.

6.1 Design Comparison of Mechanisms

In this section, we compare how the designs of the different mechanisms of expressing logical communication parallelism compare against each other with respect to two key metrics: ease of use (which measures the productivity of domain scientists), and applicability to different MPI operations (which measures the scope of the designs).

6.1.1 Point-to-Point Operations

Design 1: Communicators for Logically Parallel Communication

Communicators are one way to express logically parallel communication for point-to-point operations in both the existing MPI-3.1 standard and the upcoming MPI-4.0 standard. They explicitly denote parallelism between operations to the MPI library since operations using different communicators cannot match with each other. During our interactions with application developers from University of Utah (stencil communication in Uintah [91] and hypre [29]), Maison de la Simulation (stencil communication in Smilei [42]), Pacific Northwest National Laboratory (graph communication in Vite [49]), and University of California, Irvine (stencil communication in Pencil [102]) we discover that the matching semantics of communicators make expressing parallelism a challenging task.

> *Concern:* Exposing logical communication parallelism with communicators is a nightmare!

Deriving the mathematical relationships of mapping communicators to threads while ensuring both the sending and receiving thread use the same communicator is not straight-

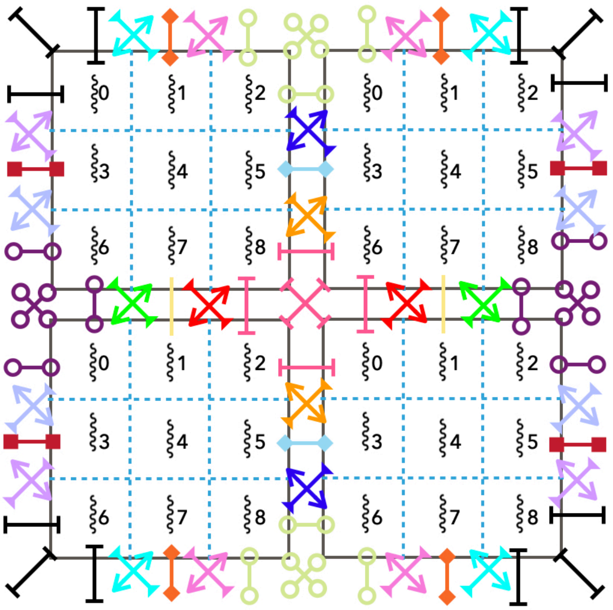


Figure 6.1: Ideal communicator usage for a 2D 9-point stencil. Each box represents a process with 9 threads. Each thread has 1 patch. Each color-shape combination represents a communicator. Numbers represent thread IDs.

forward. The complexity only increases with the increase in dimensionality. Additionally, the constraint of using the least possible number of communicators adds to this complexity. This constraint is an important one since the number of network hardware contexts is limited (see Chapter 3). By creating more communicators than there are available network resources, multiple communicators would collide on the same network resource and lead to serialization of communication underneath even with exposed logically parallel communication.

Exposing parallelism with communicators is cumbersome. Consider a static (communication pattern of each thread is fixed) 2D 9-point stencil for example. Figure 6.1 shows the ideal communicator usage—minimum number of communicators with all of the available parallelism exposed—for such a communication pattern. Each color-shape combination for a line represents a distinct communicator. 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 logically independent. The threads on a corner, however, use a single communicator for all directions since their operations for the different directions occur serially. Furthermore, the mapping of communicators to threads is not the same on each process. For example, thread 7 of the bottom-left process in Figure 6.1 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 thread 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. Doing so with a logical relation that is programmable was a hard task for the multiple application developers that we collaborated with.

Exposing parallelism with communicators is not intuitive. To elaborate on this point, we will use the stencil communication pattern as an example. The intuitive approaches that multiple application developers (those of Uintah, Smilei and Pencil) initially thought of using did not expose all of the available parallelism. In the 2D example, their intuition was 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. While such a communicator usage is correct, it exposes only half of the available parallelism. The communication of adjacent threads on an edge in Figure 6.1 occur in parallel but the operations of threads on opposite edges use the same communicator. For example, in Figure 6.1, thread 1's send operation uses communicator 1, which thread 7 also uses for its receive operations.

Mismatch in expected mapping to the underlying network parallelism. Even if the domain scientist achieves the ideal communicator usage, the number of communicators required can be much higher than the number of underlying network resources. As we saw with the hypre library in Section 5.3.1, the ideal number of communicators required was over 5× that of the number of network hardware resources. The domain scientist is expressing all of the available logical communication parallelism and expects each communicator to map to a distinct network resource, but the observed performance benefit may not be as expected because of contention on the limited number of network resources. Furthermore, we learned that domain scientists typically do not think of communicators as a means to express logically parallel communication. They have historically viewed them as groups of processes or as a means to isolate matching of messages. That it can double up as a means to express parallelism happens to be a corollary of the definition of a communicator. The multiple purposes of a communicator can lead to a mismatch in expected mapping to the underlying network parallelism. For example, an application can create a set of communicators for the purposes of grouping different processes together and later use communicators to express parallelism. The MPI library underneath cannot differentiate between the two and could end up allocating the underlying network resources to the communicators used for grouping different sets of processes, leaving lesser network resources to map to for logical-parallelism-oriented communicators. MPI libraries, however, can prevent such mismatch in expected mapping by introducing hints that allows 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.

Limiting semantics for irregular and dynamic communication patterns. The matching semantics of communicators can sometimes prevent the user from exposing all of the available parallelism in an application (e.g., the communication pattern of Legion applications in Section 5.3.3). The root cause of this limitation is that communicators constraint the task of expressing parallelism with matching semantics. So, if an application is to express parallelism with communicators it must ensure that the sending and receiving threads use the same communicator. This matching constraint can be limiting for applications with dynamic communication patterns such as those of graph and adaptive mesh refinement applications where the communication neighborhood of a thread can change over time, and for applications with irregular communication patterns where threads are forced to use the same communicator.

While using separate communicators for logically parallel communication explicitly informs the MPI library which operations are independent, doing so can be a hard task for the domain scientist. The discussion above indicates that communicators do not uniformly apply to all operations; in some communication patterns, they prevent the user from exposing the available logical parallelism. Where they are able to theoretically express parallelism, however, a steep learning curve exists.

Design 2: Tags and Ranks for Logically Parallel Communication with MPI-4.0

Through the use of new Info hints that have been voted into the draft MPI-4.0 standard, domain scientists can use tags and ranks instead of communicators to express logically parallel point-to-point communication for applications that do not use certain MPI semantics.

Existing MPI+threads applications that use `MPI_THREAD_MULTIPLE` typically already encode thread IDs into the tags of parallel MPI operations to distinguish operations that target different threads on the same target process, indicating that domain scientists intuitively think of tags as a means of expressing logical parallelism. The approach of using tags would be the one that requires the least amount of changes to existing applications. The changes would only be in the form of creating a new communicator with Info hints that relax the appropriate MPI semantics not needed by the application.

Intricate use of tags. Depending on the MPI implementation, using tags can be more flexible than communicators. Even though tags have the same matching constraints that communicators have, tags can provide more information that what communicators do. For example, for the MPICH implementation that features multiple virtual communication interfaces (VCIs), users can encode both the local VCI information and the remote VCI information into a tag, which provides all the flexibility that is needed for multi-VCI communication. However, ensuring that such information in the tags maps optimally to the underlying VCIs requires the domain scientist to intricately use Info hints to negotiate with the MPI library which bits of the tag space represent what kind of information, and, more important, how (e.g., hash function) the library should use the information. Consider the hypre application that encodes the IDs of the sending and receiving threads into the tag along with other application-related information. If the MPI library does not know which bits of the tag encode communication parallelism information, then an application is at the mercy of how the MPI library hashes the tags into the multiple VCIs allocated to the communicator. Achieving the best mapping to VCIs requires hypre to inform the MPI library 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 send operations, the MPI library can then 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 a use of tags requires Info hints that would be specific to an MPI implementation.

Limited by existing use of tags. Existing applications typically already use MPI tags for application-related information. Since the number of bits in an MPI tag are limited, depending on how many bits an existing application is already using, the app may or may not be able to encode further parallelism information into this tag. Encoding parallelism information into a lesser than ideal number of bits would not expose the maximum amount of logical parallelism. We would like to note that we have not yet encountered an application where such a problem exists.

Design 3: User-Visible Endpoints

User-visible endpoints provide a flexible interface to control multithreaded point-to-point communication. They allow the user to explicitly specify which local endpoint (through the communicator handle) to use to issue an operation on and which remote endpoint (through the rank of the endpoint) the operation needs to target. Endpoints combat the various concerns that are associated with exposing communication parallelism with communicators and tags.

Endpoints are intuitive. Given their flexible interface, endpoints are an easier alternative to express parallelism even for communication patterns such as a 3D 27-point stencil. They are more intuitive to use than communicators because application developers are already familiar with the semantics of traditional MPI ranks. By creating as many endpoints there are as threads, application developers express communication parallelism by simply communicating between endpoints as they would do for MPI ranks in MPI everywhere programming.

(Almost) No mismatch in expected mapping to network parallelism. Since the concept of endpoints is different from that of communicators, application developers can continue to think of communicators as groups of processes or as a means of isolating matching of messages. By creating a communicator with endpoints, they inform the MPI library that the new communicator being created is for the purposes of exposing logical parallelism, just like the approach of using hints with communicators. Hence, the possibility of a mismatch in expected mapping is significantly lesser with user-visible endpoints than with existing MPI mechanisms. The potential for a mismatch in expected mapping is not completely eliminated with user-visible endpoints because domain scientists could create more endpoints than the number of underlying network resources. In such a scenario, some of the distinct endpoints would contend on the underlying network resources. Although we have not come across an application that would result in such a scenario, even if an application's communication pattern requires more endpoints than the number of network resources, the problem would hold with other designs as well since endpoints reflect the upper bound with respect to expressing the available parallelism.

Endpoints separate matching and parallelism information. Unlike existing MPI mechanisms, user-visible endpoints separate the task of expressing parallelism between operations from the task of matching operations. Thus, using endpoints is straightforward for irregular communication patterns, such as those of Legion applications, since it decouples parallelism information from matching information, allowing users to flexibly express parallelism within a communicator. Similarly, user-visible endpoints do not redefine the purposes of existing MPI mechanisms such as tags. Hence, if an application is already using a large portion of MPI tags, it does not need to worry about compromising its use of tags for the purposes of expressing parallelism unlike the case with tag-based communication parallelism. Furthermore, the endpoints mechanism 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.

The concern with user-visible endpoints is that it introduces new concepts to the user. But using existing MPI mechanisms overload the definitions of existing concepts such as communicators and tags, requiring domain scientists to be aware of the concept of VCIs. Hence, the concern of introducing new concepts with user-visible endpoints also holds for existing MPI mechanisms. Unlike existing MPI mechanisms which require intricate negotiation with the MPI library, user-visible endpoints provide a flexible, straightforward interface for domain scientist to express parallelism in a way that they are already familiar with.

6.1.2 RMA Operations

Existing MPI mechanisms (windows) are sufficient to explicitly expose logically parallel communication in cases involving non-atomic operations (e.g., `MPI_Get`), but are not sufficient for cases that use atomic operations (e.g., `MPI_Accumulate`). Unlike point-to-point operations where the concerns with existing MPI mechanisms include both complexity of use and limiting semantics, the concerns with existing MPI mechanisms for RMA operations are only those of limiting semantics. Given the one-sided nature of RMA operations with no matching semantics, existing MPI mechanisms and user-visible endpoints are equally straightforward to use.

Limiting semantics of existing MPI mechanisms for atomic operations. The approach of using existing MPI mechanisms to expose logical parallelism constraints the parallelism information with the atomicity semantics of a window. This constraint limits the domain scientist from being able to explicitly expose logically parallel atomic operations in a single window even when the application does not need the atomic operations to be ordered. Consider the communication pattern of NWChem's block-sparse matrix multiplication (see Section 4.6.3). The `MPI_Accumulate` operations in a multithreaded process are forced to use a single window to maintain atomicity. Even though the parallel `MPI_Accumulate` operations are logically independent, the domain scientist has no way to expose this parallelism. The best they can do is to hint the relaxation of MPI's ordering constraint and rely on the MPI library to issue operations in parallel. Without additional information from the application, however, any of the automatic mapping policies of the MPI library will suffer from collisions, preventing the user from achieving the optimal parallel issue of operations.

With user-visible endpoints, on the other hand, the application can use multiple endpoints within a single window and expose logically parallel communication while simultaneously maintaining atomicity. User-visible endpoints decouple the task of exposing parallelism from the purposes (e.g., atomicity) of existing MPI mechanisms.

A concern about user-visible endpoints that came up during our collaboration with developers of WOMBAT [72] is the misconception about endpoints being direct handles to network resources.

> *Concern:* Endpoints is a way for the MPI library to dump its job on the end user a.k.a. domain scientist. I don't know how many network endpoints there are. So, how many should I create?

This concern holds not just for RMA operations but also for point-to-point operations. It is based on the preconceived notion that endpoints are handles to network resources. More important, we learned through our collaborations that such a concern is commonplace in the MPI community. One explanation for this concern is that it stems from 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 furthered the misunderstanding about user-visible endpoints.

User-visible endpoints are *not* handles to network resources, rather they are a means for an application to flexibly express logical parallelism. This task of expressing communication parallelism is separate from the task of the MPI library to map the exposed parallelism with user-visible endpoints to the underlying parallel network resources. To answer the question posed in the *concern*, applications should create as many endpoints as there are streams of logically parallel communication. The MPI library can then funnel the streams of logically parallel communication on distinct network hardware contexts depending upon the availability of network resources.

6.1.3 Collectives

The existing MPI semantics do not allow multiple threads to participate in the collective of a single communicator. So, with a single multithreaded process per node, applications may partition collective data of a process across threads and issue parallel collectives on the different segments of the data using a distinct communicator for each thread.

> *Concern:* Does that mean I need to handle the intranode portion of a collective?

Depending upon the nature of the destination buffer of a collective, users may need to perform the intranode portion of the collective, as we see in Figure 6.2. For example, if the count of the destination buffer of an allreduce operation is 1, then the application needs to perform a reduction step after all threads have completed the internode part of the allreduce. With user-visible endpoints, however, the domain scientist need not think of a collective as two steps. Endpoints allow threads to participate in the collective of the same communicator through different endpoints. The MPI library will then conduct both the internode and intranode part of a collective before returning from the operation.

Although a performance comparison between the two approaches remains to be investigated, from a design perspective, the user-visible endpoints interface is better because it does not force the user to manually handle the intranode portion of an MPI collective.

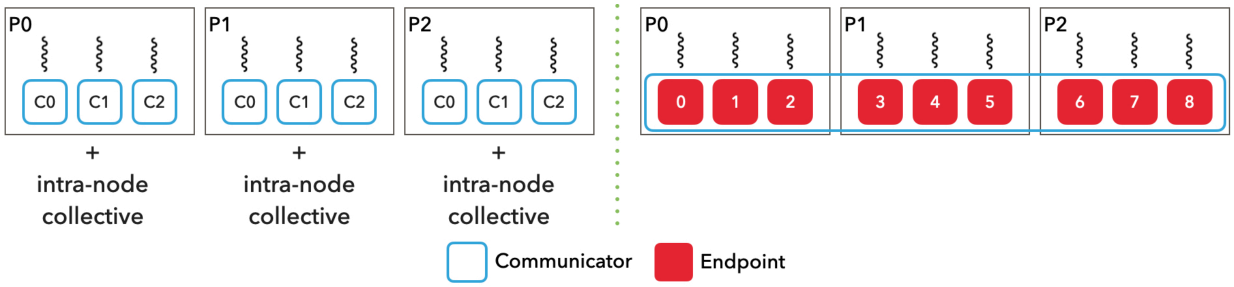


Figure 6.2: Existing mechanisms vs. user-visible endpoints for collective communication.

6.1.4 Portability

As we saw in Section 6.1.1, achieving the same level of control and flexibility of the user-visible endpoints interface with existing MPI mechanisms requires a careful use of Info hints. Since the concept of a VCI is not part of the MPI standard, it is likely that different MPI implementations will support communicator and tag-based mechanisms of exposing parallelism in different ways. The Info hints to control the allocation of and mapping to internal communication channels will be specific to an MPI implementation.

> *Concern:* To ensure portability of my code, I will need to learn about the hints available for each MPI implementation I want to use.

Since HPC application developers are geared towards performance-oriented codes, they are expected 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.

The user-visible endpoints effort, on the other hand, encourages standardizing the concept of endpoints through new APIs. With a standardized interface that provides all the information that an MPI library needs to optimally map logically parallel communication to the underlying network parallelism, applications would be portable across MPI implementations.

6.2 Concluding Remarks

Table 6.1 shows a summary of the options available for applications with existing MPI mechanisms and user-visible endpoints for the different types of MPI communication.

Table 6.1: Summary of design choices to expose logically parallel communication.

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

We note that the user has to be aware of a multitude of options when exposing parallelism through existing MPI mechanisms, and that they need to know which mechanisms become available through hints that relax different semantics. These multitude of options indicate a steep learning curve even with existing MPI mechanisms. With user-visible endpoints, on the other hand, users need to be aware of only one option: endpoints, which apply uniformly to all types of MPI communication.

The only concern that we came across with user-visible endpoints was a misunderstanding among domain experts about what endpoints represent. One of the reasons for this has been that the MPI Endpoints proposal contains terminology that is oriented towards the community of MPI library developers. But 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. Hence, to resonate better with domain experts, we propose a new name to rebrand[1] the proposal: *MPI Rankpoints* since it emphasizes the fact that users can create multiple MPI ranks within a process. The goal is to change the current mindset of domain scientists 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. It does require 2 new API extensions but it avoids the bleak consequences of using existing MPI mechanisms which may result in non-portable applications written by misguided domain scientists.

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

Chapter 7

Concluding Remarks

7.1 Summary

This dissertation uniquely bridges two ends of the HPC software stack—application developers and MPI library developers—to achieve a common goal: exascalable communication with MPI+threads application. We summarize the main contributions of this dissertation below.

Low-level communication studies. We study the factors governing the communication performance of a system in Chapter 2 where we present a detailed breakdown of an Arm-based server and share measurement methodology so that researchers and engineers can investigate the bottlenecks in the communication performance of the systems of their interest. In Chapter 3 we studied the communication capabilities of modern network hardware with respect to multithreaded communication. Through a resource-sharing analysis, we learned that a multithreaded environment can achieve the same performance as that of a multiprocess environment but just using a third of the resources.

Fast middleware. Using the lessons from our low-level studies, in Chapter 4, we developed a fast MPI+threads library that is capable of mapping logically parallel communication to the underlying network parallelism. This work is the first to achieve scaling communication throughput similar to that of MPI everywhere for both point-to-point and RMA operations without sacrificing correctness for performance within the bounds of the existing MPI-3.1 standard.

Unlocking MPI+threads applications while listening to domain scientists. Using the fast MPI+threads library, in Chapter 5, we showcase how domain scientists can dissolve the multithreaded communication bottlenecks in their applications by exposing logically parallel communication. MPI+threads applications can now perform upto 2× faster than the traditional MPI everywhere applications, a feat that significantly boosts scientific productivity and breaks the decade-long trend of applications observing slower performance with MPI+threads. With our work, applications can achieve both the scaling benefits of MPI+threads and a higher performance over MPI everywhere. While we compare the performance differences of the different mechanisms of exposing communication parallelism, we also compare the designs of the different mechanisms from a usability perspective. In Chapter 6, we argue that user-visible endpoints are a flexible, intuitive and straightforward interface that promotes the development of portable applications compared to existing MPI mechanisms.

7.2 Future directions

All the upcoming US exascale supercomputers—El Capitan, Frontier, and Aurora—will feature multiple GPUs per node. The research in this dissertation is a critical precursor for heterogeneous computing environments where the preferred programming model of choice is MPI+threads+GPU—MPI+threads to efficiently program modern processors and GPU programming model (e.g., CUDA) to utilize the acceleration capabilities of GPUs. In such applications, one way to drive GPU-to-GPU communication would be through a distinct thread per GPU. Modern GPU-based distributed applications, however, use multiple processes to drive the multiple GPUs on the node [70, 63, 69, 28] because the communication performance of MPI+threads has been dismal hitherto. By dissolving the multithreaded communication bottleneck, our work opens up the potential for applications to be both efficient and performant with the ideal MPI+threads+GPU programming model.

7.2.1 Studying different ways of GPU-to-GPU communication

GPU-to-GPU communication can be driven either by the processor or by the GPU itself. In the case of processor-driven communication, either distinct processes or distinct threads would be driving communication at boundaries of GPU kernel execution. Given the research in this dissertation, multiple threads would be able to drive the communication of multiple GPUs as fast as multiple processes do. However, scaling communication performance with threads driving GPUs remains to be demonstrated; the state of the art uses a distinct process to drive each GPU [70, 63, 69, 28].

GPU-initiated communication prevents the need for waiting for the GPU kernel to finish executing before the dedicated CPU core initiates the communication of the GPU-computed data [53, 81]. One of the key challenges for such an approach, however, is the slow serial throughput of the GPU threads. The slow serial execution can limit the injection rate of messages into the network. Given the massive parallelism available on a GPU, however, it remains to be seen at what amount of parallelism can the aggregate throughput from GPU-initiated communication outperform the CPU-driven communication at the kernel

7.2.2 Logically parallel communication for deep learning

The primary influencer of modern accelerator-based architectures has been distributed deep learning [85]. As is the case with scientific applications, distributed deep learning also suffers from communication bottlenecks at the limits of strong scaling. Communicating gradients in deep learning is a major bottleneck for DNN training at scale [104, 65]. Furthermore, the algorithms for distributed deep learning are continuously evolving. These include synchronous allreduce-style algorithms and asynchronous parameter-server-style algorithms. The impact of utilizing network parallelism for distributed deep learning workloads in the different kinds of algorithms remains to be investigated. Such a work requires studying the infrastructures of distributed deep learning frameworks such as Horovod [92] and PyTorch [77] that use MPI backends.

7.2.3 User-friendly abstractions to use existing MPI mechanisms

In Chapter 5, we learned that existing MPI mechanisms perform as well as the upper-bound set by user-visible endpoints. But, in Chapter 6, we learned that domain scientists are concerned about the complexity and portability issues when using existing MPI mechanisms to express logical communication parallelism. One way to combat the usability concerns of existing MPI mechanisms would be through an abstraction layer that allows applications to express parallelism in a manner that domain scientists are familiar with. The abstraction layer would then internally use existing MPI mechanisms along with hints specific to an MPI implementation to optimally expose the communication parallelism information to the MPI library. Designing such an abstraction layer is challenging since the designs of the abstraction would need to generally apply to any arbitrary communication pattern.

Bibliography

[1] BSPMM mini-app. https://github.com/rzambre/bspmm.

[2] EBMS mini-app. https://github.com/ANL-CESAR/EBMS.

[3] Extended EBMS mini-app. https://github.com/rzambre/ebms.

[4] Functionality of BlueFlame registers 2 and 3. https://www.spinics.net/lists/linux-rdma/msg61591.html.

[5] IB Trade Association Architecture Specification. https://www.infinibandta.org/ibta-specification/.

[6] 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.

[7] 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.

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

[9] Mellanox PRM. http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.

[10] MPI-4.0 Draft Report. https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf.

[11] MPI Endpoints. https://github.com/mpi-forum/mpi-issues/issues/56.

[12] Open Fabrics Enterprise Distribution (OFED) Performance Tests. https://github.com/linux-rdma/perftest.

[13] Page Att. Table. https://www.kernel.org/doc/Documentation/x86/pat.txt.

[14] PMU tools. https://github.com/andikleen/pmu-tools.

[15] RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core.

[16] Semantics of Point-to-Point Communication. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node58.htm.

[17] Shared AV table in OFI/PSM2. https://github.com/ofiwg/libfabric/issues/5080.

[18] Team - WOMBAT & CosmoPlasma. https://wombatcode.org/people.

[19] Teledyne lecroy summit t3-16 analyzer.

[20] TOP500 List Statistics. https://www.top500.org/statistics/list/.

[21] WOMBAT-public. https://bitbucket.org/pmendygral/wombat-public/src/master/.

[22] OSU Micro-Benchmarks 5.6.1, 2019.

[23] UCS profiling, 2019.

[24] Top 500 55th edition highlights, 2020.

[25] Y. Ajima et al. The tofu interconnect d. In *2018 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*, pages 646–654. IEEE, 2018.

[26] 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*, pages 367–379. ACM, 2019.

[27] 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.

[28] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205, 2017.

[29] 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*, pages 261–279. Springer, 2012.

[30] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. Pmi: A scalable parallel process-management interface for extreme-scale systems. In *European MPI Users' Group Meeting*, pages 31–41. Springer, 2010.

[31] 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*, pages 120–129. Springer, 2008.

[32] 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.

[33] 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.

[34] M. Berzins. Status of release of the uintah computational framework. *Scientific Computing and Imaging Institute, Tech. Rep. UUSCI-2012-001*, 2012.

[35] 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*, 38(5):S101–S122, 2016.

[36] S. Bhoja et al. Fec codes for 400 gbps 802.3 bs. *IEEE P802. 3bs*, 400, 2014.

[37] N. L. Binkert et al. Integrated network interfaces for high-bandwidth tcp/ip. *ACM Sigplan Not.*, 41(11):315–324, 2006.

[38] 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.

[39] H. Casanova et al. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[40] G. Casey. Gen-z: High-performance interconnect for the data-centric future, Mar. 2018.

[41] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[42] 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.

[43] 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.

[44] 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.

[45] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[46] 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*, page 04207. EDP Sciences, 2014.

[47] P. F. Fischer. Scaling limits for pde-based simulation. In *22nd AIAA Computational Fluid Dynamics Conference*, page 3049, 2015.

[48] A. Ghiasi et al. Investigation of pam-4/6/8 signaling and fec for 100 gb/s serial transmission. *IEEE P802. 3bm*, 40, 2012.

[49] 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.

[50] 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)*, pages 36–39. IEEE, 2019.

[51] 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.

[52] 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*, pages 34–39. IEEE, 2015.

[53] 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.

[54] 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*, HPDC '19, page 109120, New York, NY, USA, 2019. Association for Computing Machinery.

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

[56] 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*, 95(12):1121–1136, 2013.

[57] 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)*, pages 36–49. IEEE, 2019.

[58] D. Holmes. Introducing endpoints into the empi4re mpi library.

[59] 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)*, volume 2, pages 652–657. IEEE, 2019.

[60] 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*, pages 212–230. Springer, 2015.

[61] 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.

[62] A. Jackson et al. Evaluating the arm ecosystem for high performance computing. In *Proc. of the Platform for Advanced Scientific Computing Conf.* ACM, 2019.

[63] D. Jacobsen, J. Thibault, and I. Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010.

[64] 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.

[65] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[66] A. Kalia et al. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conf. ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[67] S. Larsen et al. Reevaluation of pio with write-combining buffers to improve i/o performance on cluster systems. In *NAS*, pages 345–346, 2015.

[68] G. Liao et al. Performance measurement of an integrated nic architecture with 10gbe. In *2009 17th IEEE Symp. on High Perf. Inter.*, pages 52–59. IEEE, 2009.

[69] Y. Liu, B. Schmidt, and D. L. Maskell. Decgpu: distributed error correction on massively parallel graphics processing units using cuda and mpi. *BMC bioinformatics*, 12(1):85, 2011.

[70] V. Lončar, L. E. Young-S, S. Škrbić, P. Muruganandam, S. K. Adhikari, and A. Balaž. Openmp, openmp/mpi, and cuda/mpi c programs for solving the time-dependent dipolar gross–pitaevskii equation. *Computer physics communications*, 209:190–196, 2016.

[71] A. Ltd. Armv8-a memory types, 2019.

[72] 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.

[73] 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*, 26(7):1388–1407, 2014.

[74] 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*, pages 1–8, 2011.

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

[76] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen. The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism. *arXiv preprint arXiv:2007.12856*, 2020.

[77] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[78] 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.

[79] J. W. L. Paul F. Fischer and S. G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[80] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[81] S. Potluri, A. Goswami, D. Rossetti, C. J. Newburn, M. G. Venkata, and N. Imam. Gpu-centric communication on nvidia gpu clusters with infiniband: A case study with openshmem. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 253–262. IEEE, 2017.

[82] B. Pourghassemi. Scalable dynamic analysis of browsers for privacy and performance. *ACM SIGMETRICS Performance Evaluation Review*, 47(3):20–23, 2020.

[83] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–23, 2019.

[84] B. Pourghassemi, A. A. Sani, and A. Chandramowlishwaran. Only relative speed matters: Virtual causal profiling.

[85] B. Pourghassemi, C. Zhang, J. H. Lee, and A. Chandramowlishwaran. On the limits of parallelizing convolutional neural networks on gpus. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 567–569, 2020.

[86] 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.

[87] K. Raffenetti et al. Why is MPI so slow?: Analyzing the fundamental limits in implementing mpi-3.1. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, page 62. ACM, 2017.

[88] 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.

[89] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith. Openmc: A state-of-the-art monte carlo code for research and development. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 06016. EDP Sciences, 2014.

[90] S. M. Rumble et al. It's time for low latency. In *HotOS*, volume 13, pages 11–11, 2011.

[91] 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.

[92] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[93] P. Shamis et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Ann. Symp. on High-Perf. Inter..*, pages 40–43. IEEE, 2015.

[94] A. Siegel, K. Smith, K. Felker, P. Romano, B. Forget, and P. Beckman. Improved cache performance in monte carlo transport calculations using energy banding. *Computer Physics Communications*, 185(4):1195–1199, 2014.

[95] 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*, pages 487–498. IEEE Press, 2014.

[96] P. Sun. 100gb/s single-lane serdes discussion. *IEEE P802.3 New Ethernet Applications Ad Hoc*, 2017.

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

[98] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. *ACM SIGPLAN Notices*, 48(10):495–514, 2013.

[99] 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.

[100] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó. Improving concurrency and asynchrony in multithreaded mpi applications using software offloading. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[101] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Comm.*, 181(9):1477–1489, 2010.

[102] H. Wang and A. Chandramowlishwaran. Pencil: A pipelined algorithm for distributed stencils. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1203–1218. IEEE Computer Society.

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

[104] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[105] 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*, pages 24–32, 2007.

[106] 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.

[107] 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*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[108] R. Zambre, M. Grodowitz, A. Chandramowlishwaran, and P. Shamis. Breaking band: A breakdown of high-performance communication. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

Appendix A

User Access Region in Mellanox Network Adapters

The User Access Region (UAR) is part of a mlx5 NIC's address space and consists of UAR pages. Different pages allow the multiple processes and threads to get isolated, protected, and independent direct access to the NIC. The UAR pages are mapped into the application's userspace during CTX creation, allowing the user to bypass the kernel and directly write to the NIC.

A mlx5 UAR page is 4 KB, and each UAR consists of four uUARs (micro UARs). Only the first two are used for user operations; we refer to them as data-path uUARs. The last two are used by the hardware for executing priority control tasks [4]. Each uUAR consists of two equally sized buffers that are written to alternatively [9]. The first eight bytes of a buffer constitute the *DoorBell* register [9]. Atomically writing eight bytes to this register rings the *DoorBell*.

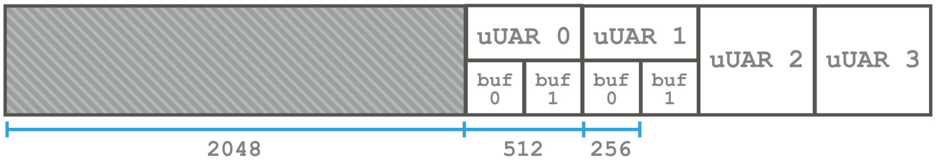


Figure A.1: 4 KB mlx5 UAR page. The last two uUARs are used by the NIC.

Appendix B

Mellanox's QP-to-uUAR Mapping Policy

When the Verbs user creates a CTX, the mlx5 driver statically allocates a discrete number of UARs. By default, it allocates 8 UARs and 16 data-path uUARs. When the user creates QPs, the `mlx5_ib` kernel module assigns a uUAR to each QP. To guide this assignment, mlx5 categorizes the statically allocated uUARs into different categories: the zeroth uUAR as *high latency*, a subset as *low latency*, and the remaining as *medium latency*. By default, mlx5 categorizes four uUARs (uUAR12-15) as low latency. Users can change this default using environment variables that allow them to control the total number of statically allocated uUARs (MLX5_TOTAL_UUARS) and categorize a subset of them (up to a maximum of all but one) to be low-latency uUARs (MLX5_NUM_LOW_LAT_UUARS).

Low-latency uUARs are called so because only one QP is assigned to such a uUAR; thus the lock on the uUAR is disabled. The medium-latency uUARs may be assigned to multiple QPs, and locks are needed to write to them. The high-latency uUAR can also be assigned to multiple QPs but it allows only atomic *DoorBells* and no *BlueFlame* writes. Hence, it is not protected by a lock.

Figure B.1 portrays mlx5's uUAR-to-QP assignment policy for an example CTX containing six static uUARs of which two are low latency (uUAR4-5). Within a CTX, the QPs are first assigned to the low-latency uUARs (QP0 and QP1). Once all the low-latency uUARs are exhausted, the driver maps the next QPs to the medium-latency uUARs in a round-robin fashion (QP2–QP6). The high-latency uUAR is assigned to QPs only when the user declares the maximum allowed number of uUARs to be low latency, in which case (not shown) all the QPs after those assigned to the low-latency uUARs will map to the zeroth uUAR.

The mlx5 driver will *dynamically allocate* a new UAR page if the user creates a thread domain (TD). Every even TD will allocate a new UAR page; every even-odd pair of TDs will map to the separate uUARs on the same UAR page, as we can see for the three TDs in Figure B.1. All the QPs in a TD will map to the uUAR associated with the TD; and since the user guarantees that all the QPs assigned to a TD will be accessed only from one thread, mlx5 disables the lock on the TD's uUAR. The maximum number of dynamically allocated UARs allowed per CTX in mlx5 is 512.

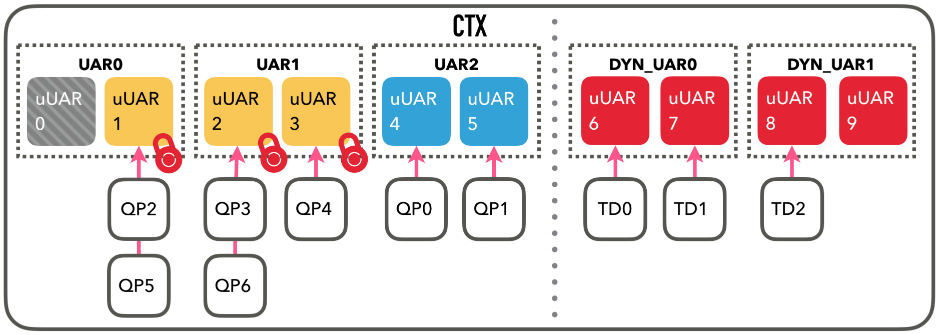


Figure B.1: Assigning seven QPs and three TDs to uUARs of a CTX containing six static uUARs, of which two are low-latency uUARs (blue). The high-latency uUAR is in grey, the medium-latency ones are in yellow, and the dynamically allocated ones are in red.