A HARDWARE AND SOFTWARE APPROACH FOR OPTIMIZING COMMUNICATION WITH DIRECT MEMORY ACCESS TECHNIQUES

Udayanga Wickramasinghe

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering,
Indiana University
January 2020

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Martin Swany, Ph.D.

Judy Qiu, Ph.D.

David Leake, Ph.D.

Roberto Gioiosa, Ph.D.

01/08/2020

For my dear parents and my loving wife,
who gave nothing but encouragement and love...

Acknowledgements

A few years back, I quit my job as a software engineer in pursuit of a Ph.D. I was forever driven by the search for new knowledge, especially in Computer Science. Thus, seemed to me that a graduate program was the best available option at the time. Little did I know that there would be many challenges during a long academic program or that my ride would have many ups and downs. I was lucky enough to meet my advisor Professor Andrew Lumsdaine, who guided me throughout this journey to explore the field of computer systems which I cherished the most. I would like to thank him for the many opportunities he presented me, allowing me to participate in cutting edge projects and research in HPC through Open Systems Lab and Center for Research in Extreme Scale Technologies (CREST). Throughout my work with him, I was astonished by his ability to analyze, articulate, and understand a broad range of problems, not limited to Computer Science. I was also pleasantly surprised and extremely grateful for the freedom he provided to all his graduate students, including me, to explore their own interests and to formulate authentic research work.

Secondly, I would like to thank Dr. Greg Bronovetsky who advised me in various capacities to help me get started and plant my foot firmly in the academic program. His deep insights and approach to technical problems taught me invaluable skills which helped me immensely in working out problems in my thesis. I would like to take this opportunity to thank Professor Sanath Jayasena, who encouraged me to pursue higher studies and also helped me immensely during admission process of Indiana University, Bloomington Ph.D. program. Third, I would like to thank all of my colleagues who helped me either directly or indirectly with my work. I was lucky enough to be surrounded by a few close friends who had similar interests in low level systems. I would especially like to thank Wathsala Vithanage, who was a systems geek himself, for the many technical discussions and arguments we had on

operating system principles and network system design. Some of the ideas that we discussed turned out to be of vital significance for my dissertation work. I would like to mention Ezra Kissel for the discussions we had and experience I gained through solving problems in the Photon project that focused on optimizing RDMA networks. I would like to acknowledge the support I received from my former colleagues – Amila Jayasekara, Saliya Ekanayake, Abhishek Kulkarni, Timur Gilmanov, Jesun Firoz and Marcin Zalewski.

Furthermore, I would like to thank my dissertation committee members Professor Martin Swany, Professor David Leake, Professor Judy Qiu, and Dr. Roberto Gioiosa for their valuable suggestions and for taking the time to participate in this process. I am grateful for the close interactions I had with Professor Martin Swany, for including me as an Instructor for his Advanced Operating Systems course and team member for a few of his research work. His expertise and guidance certainly contributed towards many aspects of this dissertation. This work would not have been possible without the resources and allocations provided to me by the Indiana University, especially with the Bigred2, Bigred2+, Dagger high performance cluster teams. I was also supported by NSF grants and the Lily Endowment in-terms of funding and resources. I also want to thank Anna Bloom, who agreed to review this dissertation with due diligence despite the tight schedule I put forward. Last, but not least, I would like to mention my heartfelt gratitude towards my dear parents and loving wife for being giant pillars of strength around me. Their unconditional love, support and encouragement helped me overcome obstacles and challenges, as well as share moments of happiness equally during this journey. Finally, I would like to thank all of my SriLankan friends, Isuru, Pavithri, Supun, Chathuri, Amila, Eroma, Milinda, Samitha, Buddhika, Dimuthu, Sachith and others in and around the Bloomington community for supporting me in various ways and above all, being great friends. I hope to see you all again in different places!

Udayanga Wickramasinghe

A HARDWARE AND SOFTWARE APPROACH FOR OPTIMIZING COMMUNICATION WITH DIRECT MEMORY ACCESS TECHNIQUES

Data Communication has two fundamental aspects. First is moving data from a locality of a logical endpoint to a desired destination. Secondly are the enabling methods of notification or receipt of data at respective logical endpoints. While optimizing communication can be trivially defined as both the transferring and notification of data by the shortest path possible, it is, practically speaking, a difficult problem because of the existence of many performance inhibitors. This thesis will show that communications performance is deeply affected by the behaviour of the underlying execution environment. High performance runtime execution environments broadly expose two types of communication, namely, two-sided and one-sided. Two-sided communication actively involves both source and target processors, but may incur additional overheads. One-sided communication defines messaging patterns where data transfer does not involve any of the processing components of the destination. Cutting edge networks interconnect technologies offer hardware assisted one-sided communication primitives such as Remote Direct Memory Access (RDMA) - A capability that minimizes remote memory access latency by bypassing the remote host processor and avoiding operating system interaction. While RDMA is extremely useful, exposing proper abstraction and application are of major concern because transfer and notification paths need to be handled separately, but cautiously.

This thesis will show that communication performance can be improved by building lightweight network software architectures that can be seamlessly integrated by both the hardware and software interface. More specifically for optimizing communication between nodes (i.e. inter-node), the thesis introduces a novel distributed dynamic remote memory allocator for RDMA called “rmalloc”, and a high performance communication channel for RDMA, called “rpipe”. The “rmalloc” and “rpipe” introduce a light weight remote memory provisioning capability along with zero copy one-sided messaging programming model called RDMA *Managed Buffers* . This work will investigate the performance characteristics of “rmalloc” with respect to network and synchronization specific characteristics and comparison with other runtime environments such as MPI, MPI3.0 RMA and state-of-the-art RDMA network libraries. A major part of this investigation revolves around rmalloc’s usage of different messaging patterns for real world parallel applications in a High Performance Computing (HPC) context.

Contents

CHAPTER 1

Introduction

"The task is not to see what has never been seen before, but to think what has never been thought before about what you see everyday..."
— Erwin Schrodinger, (1887-1961)

A significant portion of parallel applications contain some form of communication routine for exchange, query, update, or rendezvous between logical processors. Therefore, reducing communication overheads by utilizing underlying hardware resources to the fullest extent possible has become paramount for achieving high performance on runtime, middleware, and operating systems. All communication mechanisms facilitate data delivery either within the same logical endpoint (a.k.a. intra-node) or between different logical endpoints (a.k.a. inter-node). Generally, a successful communication channel is formed if two factors are met. First, it allows for an exchange of data from one logical memory address to another. Second, a certain agreement between the endpoints is made upon delivery and receipt of data. Today, high bandwidth asynchronous memory controllers (i.e. DMA), fast memory/cache hierarchies, programmable MMU, and network on chip [**23**] (NoC) devices have taken precedence over traditional IO for providing extremely efficient intra-node data access. Remote Direct Memory Access (RDMA – also loosely known as RMA) is a popular technology that has been widely utilized for facilitating inter-node communication. With RDMA, modern interconnect fabrics can provide direct access to remote peer memory with much faster speeds and accessing remote-side memory buffers with little or no CPU intervention [**122**] (a.k.a kernel bypass). Minimal congestion (or congestion free) routing coupled with RDMA have contributed significantly to the scalable performance of inter-node communication via high throughput, low latency network interconnect fabrics such as Intel, Cray, Mellanox, and

Cisco. While hardware technologies have advanced, increasing evidence suggests that software libraries and middle-ware have evolved at a much slower pace to meet the demands of parallel applications, to say nothing of efficiently programming the underlying hardware.

Many data parallel applications have relied on message passing interfaces such as MPI for communication. However, there are a number of problems in traditional message passing. MPI's distributed memory abstraction forces developers to clearly identify the information that is local to a core or remote processor, which encourages data locality. Furthermore, MPI separates the portions of the application devoted to computation and communication. However, traditional implementations of MPI have faced performance constraints on modern shared memory hardware, because their process-based design required unnecessary copying to communicate among MPI ranks on the same node. Attacking the "shared memory" problem by introducing address spaces that will be shared between MPI ranks or threads have not lead to satisfactory solution [**86, 92, 106**] – resource contention being a major performance degrading factor. MPI message passing has also exposed certain issues in inter-node data transfer. While RDMA network operations are utilized by MPI for direct zero copy transfer, its messaging model may enforce a per message fixed cost for synchronization phases known as buffer "registration", "pinning", and "exchange". Additionally, two-sided [1] models, such as MPI, may also generate a message matching overhead, which can ultimately weigh down advantages of RDMA. To mitigate these limitations, MPI introduced one-sided message passing, MPI-2.2 and MPI-3.0. [**37**] RMA – a major upgrade from the traditional MPI programming model which has proven more performance friendly towards RMA-capable network fabrics. However, in terms of programming models, MPI-2.2/3.0 RMA need more discipline and a deeper knowledge of the platform and runtime system implementation in order to get around what many consider weaknesses in coarser granular bulk synchronization (active case), as well as strict memory consistency constraints and portability issues (passive case) [**111, 36, 123**].

[1] Where both sender and receiver are aware of the message

Alternatively, native one-sided and higher level programming models, such as PGAS (i.e. UPC, HPX-5, OpenSHMEM), deviate from the data parallel model and instead focus on programs written for readily accessible global memory (i.e. access memory whatever/whenever). They make use of transport level library solutions (i.e. GASNet [**11**], UCX [**101**] Gravel [**33**], Photon [**73**], etc) that support a fine granular remote memory access model for communication. These libraries offer more control over RDMA network operations and provide notification events that facilitate RDMA operations by indexing interconnecting specific data events on special hardware queues called completion queues (CQ). While such middle-ware libraries work closer to the fabric and thus, may provide maximum utilization of the network adapter/hardware, their target audience is runtime system developers who will in turn use them as tools to build higher level communication primitives and programming paradigms. One of the overarching issues of RDMA is efficient synchronization handling. However, while native RDMA protocols decouple communication from synchronization, enforcing correctness in synchronization for transport protocol is a non trivial problem. In order to solve the "synchronization" problem, an approach like busy waiting over communication links or network queues may become counterproductive. Many runtime systems and libraries have adopted sophisticated flow control protocols [**121, 53, 66, 79**] such as credit flow control, counting barriers, or bulk synchronization (i.e. MPI-3.0). Although bridging the gap between application and the underlying network fabric via a high-performance low-level middle-ware interface is highly desirable from a programmer's perspective, building such sophistication directly into a parallel application is questionable. Therefore, next generation message passing middle-ware and libraries may need to address these concerns regarding mitigating overheads towards the network layer with the abstraction level being exposed to user-land applications, as well as optimizing for maximum leverage of underlying hardware.

Performance losses may occur due to various causes in multiple layers of a software stack. It is often the case that it is difficult to single out one specific reason for performance degradation. Multiple factors may contribute to the observed behaviour. In this thesis, I broadly categorize communication bottlenecks into three main areas: a) Performance losses due to synchronization cost; b) Losses due to impedance mismatch in hardware and software abstractions, programming paradigms, and execution; c) Performance losses due to sub-optimal network implementations – use of heavyweight routines in critical path, unwanted copies, etc.

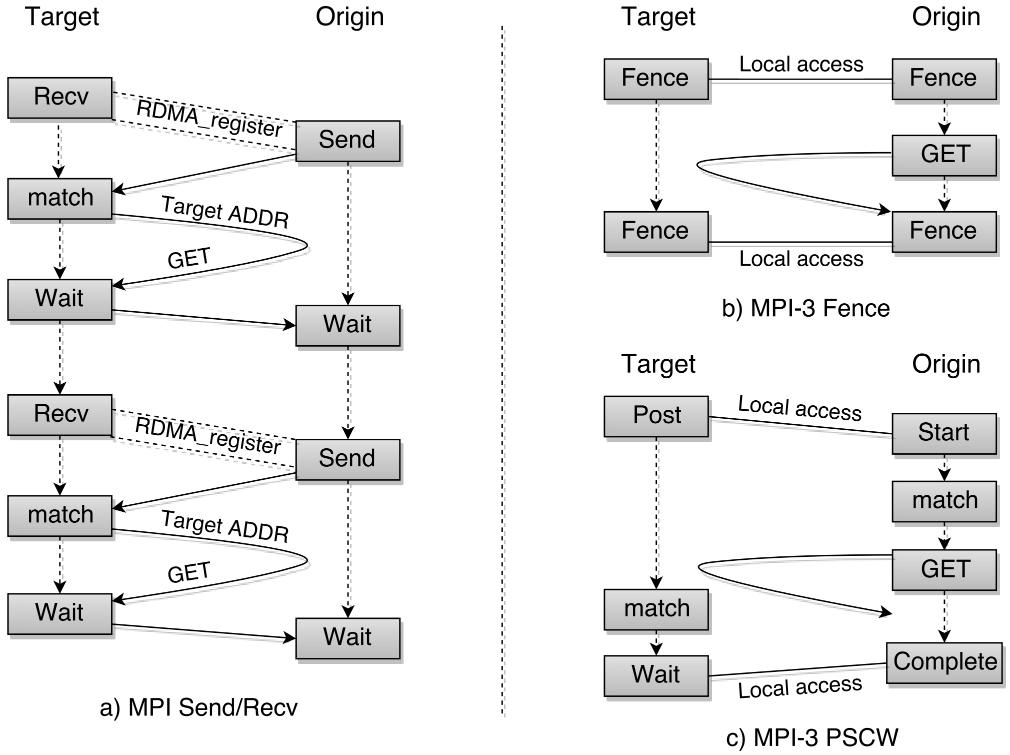


Figure 1.1. **MPI vs MPI-3.0(RDMA)**

1.1. Synchronizing Endpoints

Many real world computation problems require coordination between the moving parts in its communication routines. Similar to signals guiding traffic in a busy junction, synchronization primitives help assist correct timing and notifications for communication between endpoints, in order to avoid undesired results such as garbage data, partial data delivery, or in some cases, deadlock between processes. Often, synchronization enforced by barriers is essential to separate programs into different phases. Synchronization via locks are equally necessary to ensure mutually exclusive access and to avoid data races. Synchronization is considered a necessary evil in parallel programs. However, wasteful or coarse grained synchronization may become detrimental to communication performance [**109, 123**]. Coarse grained synchronization, such as redundant barriers, are often found in rendezvous points (i.e. handshakes) within MPI. Many layers of software abstraction may also contribute to synchronization issues when some of the rendezvous points are hidden to the user. Fine-tuning middle-ware systems for RDMA is a challenge, due to the many issues surrounding synchronization. Integrating RDMA techniques for message passing has thus become an area of significant research [**33, 50, 13, 82, 62**]. In MPI, two-sided communication semantics may often inhibit possible performance gains by RDMA. An example is shown in Figure 1.1-a. To transfer medium and large sized data[2], an association must first be established between source and destination endpoints, and the appropriate memory buffers must be registered with the network adapter (and/or "pinned down", so that operating system page table mappings would not be modified until the transaction is complete). Finally, buffers are exchanged securely so that MPI can initiate the transfer. Message passing incurs an additional message matching overhead [**43, 96**] imposed by the MPI specification.

In contrast, MPI-2.2/3.0 may reduce buffer registration and pinning costs by using remote memory windows [**54, 37**]. In MPI-3.0, active RMA (Figure 1.1-b and Figure 1.1-c), creates an exposure epoch that ameliorates synchronization costs by bulk execution of communication operations. However, unless network operations are batched together, barrier/fence synchronization, enforced ordering guarantees, (accumulate and atomics), and pairwise RDMA (PSCW) matching phases may still sustain some of the bottlenecks described above. For Irregular RDMA, coarse synchronization by aggregating completion events can be disadvantageous owing to the lack of fine-grained control that negatively impacts applications, thus reducing the ability to overlap communication and computation. Enforcing synchronization is often confused and complicated with passive targets [**9**] (i.e. `MPI_Win_lock` and `MPI_Win_unlock` and `Win_flush`). Causality of local and remote completion events

[2] for small messages MPI would use a more efficient eager protocol

may be masked by the specification details (i.e. supported consistency model) and application performance could depend entirely on the MPI implementation and level of network support available (e.g. obscure remote agents to check the progress of the network when RDMA support is unavailable). The runtime execution environments in the partitioned global address space (PGAS) attempt to address these issues via the programming model itself, making it insusceptible to some of the synchronization bottlnecks in the RMA model. However, PGAS languages (i.e. UPC, Charm++, OpenSHMEM, ARMCI) that implement MPI network backends may suffer from poor communication performance [**105, 36, 28**], due to the overheads of bulk synchronization primitives. Moreover, PGAS environment may carry a heavy overhead when dynamically allocating and de-allocating RDMA memory. For instance, PGAS environments like SHMEM initializes RDMA address space by allocating cluster wide RDMA memory at startup. However, if requirements change dynamically, buffer exchange phases for reallocation then becomes an issue with increasing coherence traffic and stalls until `realloc()` updates are completed.

We anticipate that fine-grained control and scalable synchronization protocols will prove to be powerful mechanisms for gaining maximum advantage of RDMA-like capabilities. However, fine-grained notified access [**13**] for synchronization is a difficult art to master. Each network hardware supports a multitude of features and thus, would vary in terms of available support for different network platform architectures. For instance, Cray network adapters impose completion queue (CQ) event constraints for each transaction type or they impose numerous alignment requirements on their network transactions, such as 8-byte alignment on synchronization and flags for Fast Memory Access (i.e. FMA) transactions in Aries which is not required for Gemini. The challenge is to provide support for scalable synchronization protocols by using the optimal notification path while the hardware characteristics vary by the transaction type, network adapter, and the platform.

1.1.1. Evaluating MPI. Common elements of the RDMA protocol such as endpoint meta data exchange and buffer synchronization protocol are generally invisible, however, they are also embedded in higher order communication routines in runtime systems such as MPI,

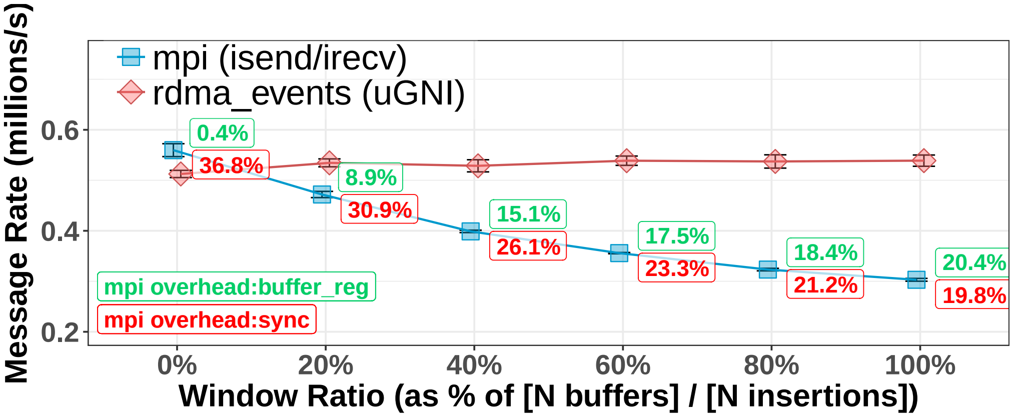


FIGURE 1.2. **Performance for RDMA zero-copy on a message window by MPI and low-level event-based RDMA for a payload size of 4KB in Cray-XC**

the de facto standard for message passing. While it is understood that software abstractions function to hide low-level details about the hardware and software interface, evidence [**96, 43**] suggests that they sometimes fail to succinctly capture the performance-critical path in user-facing interfaces. For example, point-to-point communication routines that use RDMA in MPI/MPI 3.0 mask costs associated with message synchronization and buffer registration on the host network adapter.

Figure 1.2 reports an example when a network zero-copy is initiated over 100,000 iterations by standard MPI routines and direct event-driven RDMA. The rate of messages decreases rapidly when the window ratio [3] is increased in MPI because the number of buffer registrations increases linearly and even gets over the relatively fixed cost involved in matching and ordering in MPI runtime. In contrast, an event-driven RDMA approach has a fixed buffer registration cost and synchronization that can be made lightweight, provided RDMA buffers and protocols are managed efficiently.

[3] Refers to the ratio of number of buffers to number of messages sent in an application. The window ratio is higher when large number of distinct RDMA buffers were utilized and lower when few distinct buffers were used.

1.2. Impedance Matching

Impedance matching is the practice of designing communication software and middleware to maximize the underlying system resources such as memory subsystems, processor, Input and output (I/O) interfaces, etc. Middleware designed for parallel applications often falls short of the maximum utilization of the underlying hardware. It is sometimes the case that a generic software model fails to adequately capture the hardware features being offered such as parallelism, locality, and fast notification paths. Other times, software needs to be tweaked extensively to match each application scenario because the programming model imposed barriers that create communication bottlenecks. The impedance mismatch between software programming models and hardware interfaces may be a detriment to performance. The symptoms of impedance mismatch can be seen most often in irregular execution – when there exists a **spatial** or **temporal** imbalance in a parallel program [4].

1.2.1. Impedence Problems in Hybrid Systems.

Hybridized paradigms, like MPI and X (i.e. MPI+X), do not always converge to optimal performance. Such systems are analogous to certain human involved experiments with hybridization in the natural world, which have not flourished due to inter-species incompatibilities (i.e. zebroids and ligers come to mind[5]) Similarly, it is questionable where or not a hybrid system is sustainable in a rapidly evolving technology context. Moreover, a number of issues related to synchronization between run time boundaries, model-imposed barriers, and various overheads in heterogeneous scheduling have hobbled these systems [**116, 17, 114, 21, 38**].

Collective communication comprising synchronization barriers are known to propagate and even amplify noise effects. The effects of noise directly impact application scalability [**64, 42, 10, 5**]. With MPI+X there is a necessary sequence of operations (local plus global) that combine to create a single compound operation. The effect of computational irregularities

[4] **Spatial** imbalance relates to load imbalance of data across the system, while **temporal** imbalance directly corresponds to noise or delays in the system relating to NUMA access, the operating system, or the network.

[5] A zebroid is a hybrid between zebra and any horse/equine family of species. A liger is a hybrid of lion and tiger species.

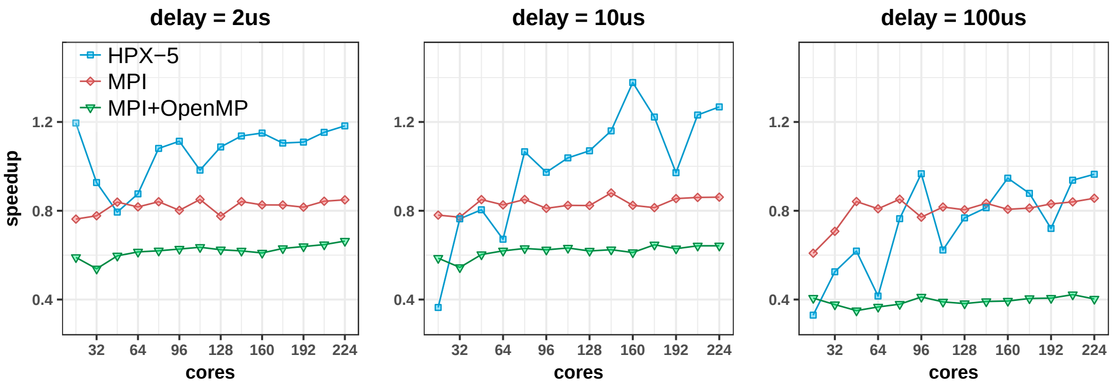
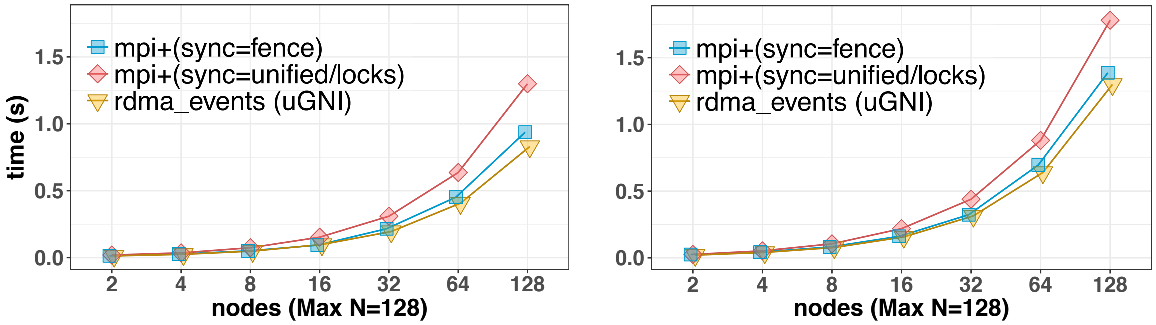


Figure 1.3. **Performance comparison of MPI, PGAS and MPI Hybrid for a global reduction – relative slowdown when different levels of noise injected in an Infiniband^{TM} cluster environment (224 cores)**

is isomorphic in relation to system, but there are also, potentially, orders of a larger magnitude. More importantly, in similar irregular settings, the synchronization at the initial phase of reduction limits overall parallelism and throughput achievable by the application. For instance, a global parallel reduction by MPI+OpenMP imposes a serialization (albeit a coarse one) of the operations in the parallel reduction – i.e., it requires a reduction of the local variables followed by a further reduction of those intermediate values. Such a coarse serialization may not appear to be detrimental and, thus, would also seem to be the best possible approach. However, if the local reductions are not well-balanced, which can happen in the case of imbalanced applications, this serialization may cause further delays [**116**] to the collective reduction operation. Figure 1.3 shows the relative speedup for an irregular[6] global reduction amongst contrasting programming paradigms. In this case, a mismatch between execution environments is evident such that, a hybrid paradigm (i.e. MPI+X) seem to be more susceptible to severe performance degradation especially during global synchronization tasks such as reduction. Thus, it is advisable to approach hybridization more diligently without compromising runtime boundaries in each participating execution environment.

[6] Irregular because temporal delays are present in the reduct input site.



(A) **function time = 4us** (B) **function time = 40us**

FIGURE 1.4. **Performance comparison of a `alltoall` in MPI-3.0 fence, fine-grained completion(locks) and event-based low-level RDMA in Cray-XC40**

1.2.2. Impedance Mismatch in RDMA. Impedance mismatches in pure RDMA one-sided programming models also expose communication performance issues. The coarse-grained nature of higher order communication routines, along with programming model constraints, inhibit parallel applications from reaching the native performance offered by the hardware. The MPI programming model for RMA allows for relaxed semantics due to lack of support for non-RDMA capable platforms. For instance, MPI-2.2/3.0 passive RMA programming requires users to explicitly flush communication channels to make sure that the network is progressing properly. Moreover, it may require some form of active involvement in the target (i.e. using locks or synchronization primitives such as waits, fences, or active agents to progress networks) in order to write portable MPI programs for TCP/IP or shared memory. Finally, to make matters worse, each MPI implementation may relax these semantics arbitrarily [**9**], thus making it difficult to effectively execute RDMA enabled applications on different environments (i.e. Intel MPI vs OpenMPI). As a consequence, unnecessary synchronization and other overheads may affect the overall communication gain on platforms that have excellent support for RDMA's true capabilities. From a programmer's perspective, MPI-3.0 RMA applications need discipline and a deeper knowledge on platform/environment to optimize performance [**111, 67, 66**]. This is an extremely difficult task considering each MPI application must be re-written for respective platforms or environments.

While one-sided communication routines in MPI 2.0/3.0 support a fine-grained, event-driven RDMA programming model to a certain extent, the semantics of the RMA memory model [**9**] have resulted in hidden synchronization bottlenecks during network progress [**36, 123**] phases. Even in its performance friendly form of relaxed consistency RDMA model[7], data parallel applications have managed to gain little ground in adopting MPI3.0, however, an increased complexity in developing portable applications, and the sub-par performance of coarse-grained programming models have been major hurdles. Figure 1.4 shows a generic *alltoall* performance with bulk (i.e. fence) and fine-grained mode (i.e. lock/unlock) in MPI3.0 and direct event-driven RDMA with varying processing times[8]. Despite the presence of extremely dense communication graphs, the native fine-grained RDMA has shown significantly less synchronization overhead (about 4 – 15%) than that of MPI3.0 RMA.

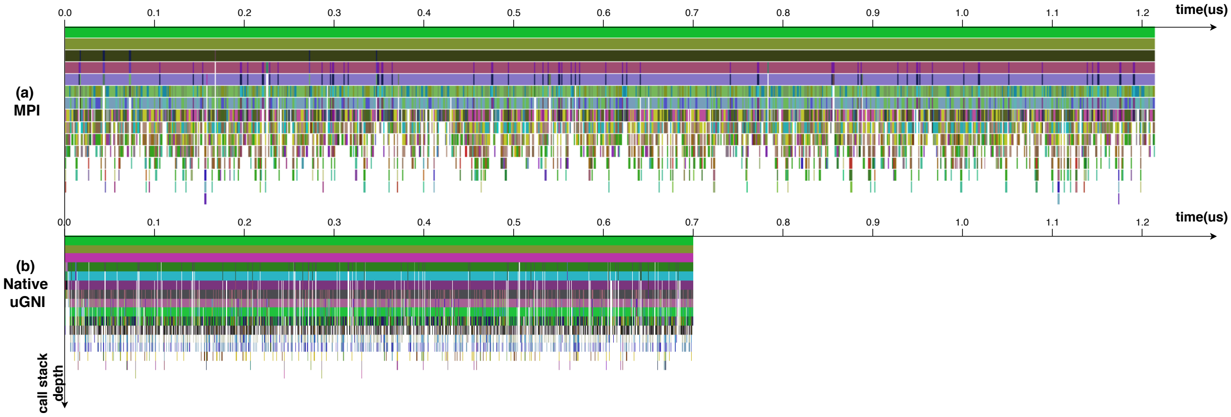


FIGURE 1.5. **A comparison of Software Stack Call-path Density in a Simple RDMA Put Benchmark for MPI RMA and the Native Transport in Cray-XC30**

1.3. Optimizing Software Implementations

Some of the challenges for developing high performance communication substrates stems from developers' inattention to optimization opportunities. The most common example

[7] In MPI3.0 RMA, relevant interfaces allow users to program in two separate memory models (*aka "unified(relaxed)"* and *"separate(strict)"*) depending on the hardware availability for direct memory coherency.

[8] Each message after *alltoall* completion was assigned a processing *function* to emulate varying computing regions of 4 us and 40 us for targets

for a sub-optimal communication routine is one that implements unnecessary copies during critical path. Other common factors [**26**] include failing to realize the hardware-software properties, such as data locality, fast path network operations, reducing frequent network operations that are too costly for overall communication, insufficient exploitation of parallelism, communication/computation overlap, and poor choice of protocols. Many a time, a higher depth of layers in a software stack may be a signature in identifying hidden bottlenecks that may become invisible to applications. An example of the difference in subroutine call-path density for a RDMA operation that is twice as slow compared to native network software(i.e. uGNI), is shown by Figure 1.5. While in both occasions the same operation is executed, MPI RMA reports a larger density of subroutines than the native transport. It is evident that when there is a higher-level software, the abstraction critical path generally becomes longer for a larger duration of time, increasing the probability of adding collateral costs.

1.3.1. Optimizing for Distributed Memory. While implementing intra-node direct memory access paths invariably benefits communication, adapting a similar approach between distributed memory nodes requires both hardware and operating system support. Modern RDMA technology offers the potential for exceptional communication latency and throughput. However, design choices including which RDMA operations to use and how to use them will significantly affect observed peak performance. Allocating memory-on-device for communication setup is often achieved via `ioctl` or a related library interface and thus, splits the address space into regular application memory (i.e. memory allocated by `malloc`, `mmap`, `memalign`, etc) and memory used by the network device. Therefore, the user does not have direct control over device managed memory by MMIO (Memory Mapped IO) or DMA (Direct Memory Access). RDMA user libraries like uGNI [**105**], DMAPP [**105**], Openfabrics [**90, 84, 56**], libfabric [**84**], UCX [**101**] facilitates this process, however, the performance and scalability of an RDMA-based communication protocol depends on several factors including the operation type, transport, optimization flags, and operation initiation method [**71, 45**].

The transaction layer packets(TLP) are transferred between device and CPU through a message bus which is a few folds slower than the native on-die interconnects. Therefore, in order to facilitate efficient communication, proper utilization of PCIe bus, network processing units are important. Many NICs support latency friendly fast network paths based on memory mapped IO (MMIO) for small messages [**14, 71**]). Each CPU store on an outbound or inbound TLP transaction is translated to PCIe IO operations and thus, communication performance greatly depends on the number of MMIO operations issued. Host CPUs implement a method called ”write combining” to make MMIOs more efficient, which batches up to a cache of line-sized stores. While MMIO writes are best used for smaller messages, for large messages DMA reads (a.k.a. Doorbell) are utilized to optimize for bandwidth. However, a DMA read almost always uses less host-to-device PCIe bandwidth than an equal-sized MMIO [**71**] – this is an important factor considering optimization for certain types of messages that exceed combining lengths. Other optimizations [**71, 53, 121**] include using doorbell methods to batch requests, compacting or shrinking payload, and overloading headers with some payload information. Furthermore, improvements to communication can be made by leveraging NIC processing units (PU) properly. Disabling some signalling events like notification events on CQs which carries a certain processing overhead by the NIC PUs and using link layer assisted synchronized network actions are some of the methods that can be utilized [**71**].

Direct implementation of low-level RDMA event based routines (i.e. libfabric, verbs, uGNI , GASNet, UCX, etc) in parallel applications is an sophisticated exercise. The complexity of this approach stems from the handling of the separate data and synchronization paths of the network hardware abstraction layer (i.e. completion events, queues, connection domains, routing modes, and endpoints). Managing the software complexity of flow control, multiplexing of resources, and optimizing in a performance portable way across platforms and various application logic may lead to user errors and sub-optimal implementation. The use of low-level RDMA substrates generally are not targeted towards the application developer, but rather, towards system and middleware developers. Thus, a high-performance solution should encapsulate sufficiently verbose abstractions for high level applications, as well as efficient transport implementations of the network subroutines and protocols involved.

CHAPTER 2

At the Crossroads of RDMA Communication Networks

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— Alan Turing, (1912-1954)

2.1. RDMA Technology

RDMA networking technology allows direct access to the memory of a remote compute node. RDMA enabled network adapters including InfiniBand, Infinipath/Omnipath, and Cray allow for exposing high data bandwidth and low-latency local area networks. Furthermore, Host Adapters with up to 800 gigabits of per-port bandwidth and ~1µs round-trip latency are currently available. However, proper utilization of the RDMA network fabric is required in order to achieve better performance and scalability. One way to think of this is to imagine handling a sports car: While a sport car comes with bigger and better engines, not to mention the transmission and raw horse power than a regular sedan, it also requires a driver skilled at handling extremely high speeds and forces. RDMA similarly demands a developers' thorough understanding of the internal mechanics and protocols that eventually transform into robust software design.

2.1.1. RDMA Hardware Design. RDMA Network Host Control Adapters (HCA) or more commonly termed, Network Interface Controllers (NIC), are devices with an intrinsic control logic for outbound and inbound data handling. RDMA enabled NIC makes use of the hardware interface in order to communicate with the driver program, which is running privileged operations for direct memory-to-memory access. Each device has a special control logic for packet routing and linking which are different from traditional TCP/IP packet handling.

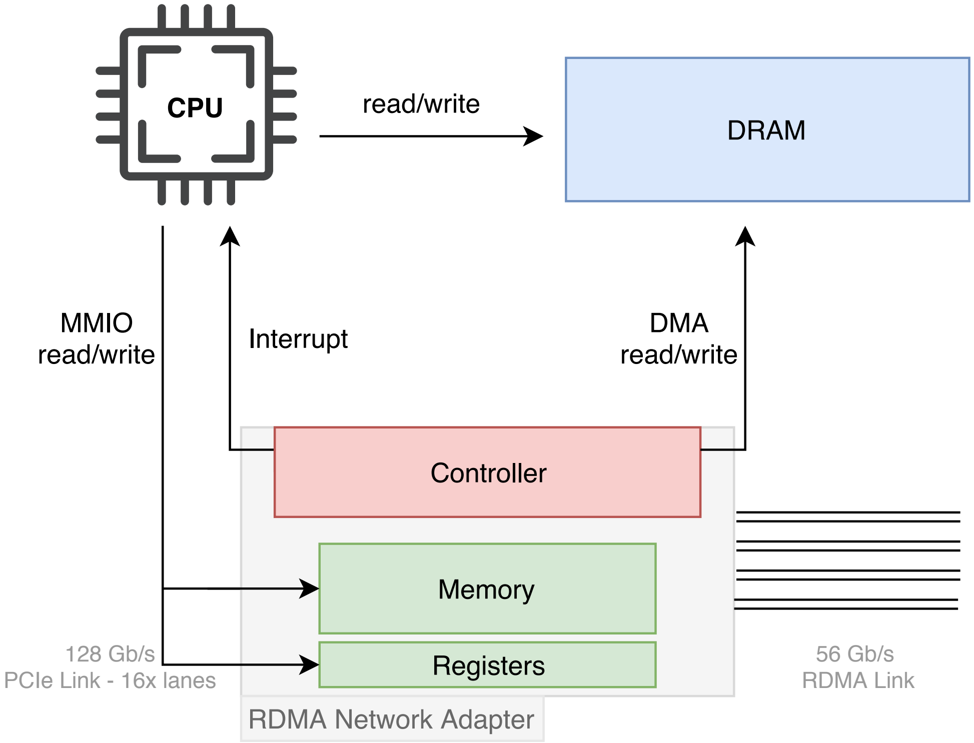


FIGURE 2.1. **Hardware Architecture for RDMA Networks**

A typical adapter also contains memory and registers built-in data movement and control between processor and the device. Current network interfaces (NIs) are predominantly discrete PCIe cards. While vendors are beginning to integrate adapters on-die [**103, 22, 2**] or socket, they still communicate with the PCIe controller using the PCIe protocol, and thus, are less powerful than discrete NICs.

2.1.2. Differences in Network Links. Figure 2.1 reports a hardware architecture that is common for most RDMA enabled NICs. Each adapter comes with device intrinsic memory for storing transient data from cpu or the device, as well as for implementing transmit(TX) and receive(RX) buffers for each logical connection. Device registers are usually small in size, varying between 8, 16, 32, and 64 bytes. Device memory is relatively larger than the registers and used to store intermediate packet data. However, like many PCIe devices, transactions on memory not only store/retrieve data, but also initiate other side effects. For

Component	Theoretical Peak Bandwidth	Type
PCIe 2.0	8 GB/s	16 lane
PCIe 3.0	16 GB/s	16 lane
PCIe 4.0	32 GB/s	16 lane
Mellanox/Infiniband	7 GB/s - 25 GB/s	HDR
Intel Omnipath	12.5 GB/s	-
Cray	10.5 GB/s	XC/Aries
Cray	12.5 GB/s	XE/Aries
Cray	25 GB/s	Slingshot

Table 2.1. **Bandwidth values for Mellanox, Intel, Cray RDMA links and standard PCIe Components in NIC(s)**

example, writing an RDMA control register may start a network transmission sequence or register an interruption for a completion event.

In general, a device has two types of data communication lanes. First is the internal data path that interfaces with both processor (cpu) and memory. Second is the external data link for communication with remote nodes. The intra-communication lanes of the device are connected by a PCIe link which may be split into multiple sub lanes. It is important to note the transmission properties of each link type displayed in Table 2.1. As shown by the data, external link throughput may always be bottle-necked by the PCIe bandwith. Thus, proper handling of PCIe transactions by software is an important consideration for RDMA efficiency. PCIe is a layered protocol, therefore, the layer headers add overhead. The amount of header overhead varies with the type of communication mode and message size involved. Each RDMA operation may also generate one of three types of PCIe transaction layer packets for reading, writing, and reading completion events [**71**]. While a single PCIe transaction is the fastest way to transfer data between the RDMA device and cpu, multiple transactions may carry additional header overhead and cpu cycle penalty.

2.1.3. MMIO vs DMA. There are several important differences between the two methods of data transfer from a CPU to a PCIe device. CPUs may write to mapped device memory (i.e. MMIO) [1] – which will initiate PCIe transaction layer packets via available lanes. A

[1] This a functionality called different names. For example, Cray names it FMA(Fast Memory Access), while Intel calls it Omnipath, and Mellanox calls it PIO (Programmed IO) Send and Blueflame

transaction may write directly to device control registers (i.e. network task entry) or the memory (i.e. a data descriptor). CPUs may use an optimization called “write combining” to avoid generating a PCIe write for each store’s instruction. PCIe devices have DMA engines and can read from or write to DRAM with DMA [2]. A device’s DMA controller may generate an interrupt when DMA reads or writes are completed. This is the mechanism used by device drivers to push completion events into user programs. DMA reads are also not restricted to cache lines, but a read response larger than the CPU’s read completion combining size is split into multiple completions. Due to the fact that MMIO generate extra header overhead than DMA, a DMA read/write always uses less host-to-device PCIe bandwidth than an equal-sized MMIO. However, because MMIO is synchronous, it is generally faster than DMA (i.e. does not have to wait for asynchronous signals). This difference in MMIO versus DMA is crucial for understanding and optimizing RDMA.

Reading/writing through device registers and memory (i.e. MMIO) will route all IO through the CPU. It utilizes CPU cycles for data transfer and also requires additional PCIe packet headers for routing the message. While this mechanism is useful for small messages to transfer packets with as minimal latency as possible, it also causes huge bottlenecks for large messages. Instead, DMA is used for more throughput friendly large(r) message transmissions. While MMIO is more or less straightforward in terms of interfacing cpu and RDMA device, DMA is more involved because of the additional setup routines. The basic approach to DMA instructs the device where to fetch or put data in memory. Once this step is completed, the device adapter will engage in data transfers to/from memory without any cpu intervention. DMA buffers are allocated in memory in terms of physical page addresses [**16, 71, 89**] [3].

Certain messaging modes for RDMA implements a pre-allocated “ring” of DMA buffers as send (i.e. TX) and receive(i.e. RX) to/from message queues. These are naturally ring structured DMA memory buffers with device owned head and tail registers pointing to the

[2] Vendors use different names for their proprietary DMA protocol – Mellanox Infiniband’s Door-Bell, Intel SDMA (Send DMA), Cray BTE (Byte Transfer Engine),etc

[3] This is similar to IDT and page table setup, but devices must know their DMA regions from the underlying operating system

start and end of a queue. A RDMA device advances head pointer to get the next valid buffer once data is copied from device to DMA memory. Similarly, the device driver may write to a tail register to provide a notification that data was successfully consumed. Eventually, the device drivers may use one of two mechanisms to detect buffer updates.

- Memory Polling - driver polls head pointer and checks for any update to its register/memory.
- Interrupts - the device injects an event into the cpu in terms of an interrupt. This *completion* event will signal the driver to start the necessary processing.

Assuming a network device is DMA enabled, a generic algorithm for handling DMA ”ring” buffer transmission and receive may look like the following: While this algorithm may be specific to a RDMA driver, it is useful to build synchronization algorithms with RDMA. This is because most modern RDMA network adapters expose full control over DMA buffer handling to userland applications without needing to use internal DMA queues.

(1) **Packet *Recv***
   (a) Device copies data into empty buffer pointed by *head* in the ring and advances *head* pointer
   (b) If driver polls then tests until head has changed; goto (d)
   (c) else device sends an interrupt and signal is then received by the driver; goto (d)
   (d) Driver consumes buffer by either copying or calling back the user application
   (e) Driver writes to the *tail* register (or *doorbell*s) when DMA buffer read has completed. Device registers this buffer as free next incoming messages.

(2) **Packet *Send***
   (a) Driver prepares data into end of queue pointed by the *tail*
   (b) The device detects that a new buffer has been added via the *tail* register (or *doorbell*)
   (c) If driver polls then tests until *head* has changed to free the buffer

(d) device sends an interrupt and signal is then received by the driver to advance the *head* pointer.

2.1.4. Network Transports. RDMA hosts communicate by exposing some hardware abstraction layer construct(s) built into the software. For example *verbs* API refers this to a *queue pair*(QP) or cray uGNI refers to an *endpoint*. Such constructs represent a secure logical connection between hosts. To establish a connection, hosts have to communicate any meta information required (i.e. network addresses, secure keys, buffer addresses, MAC, etc) with each other. RDMA transports encapsulate many connections and are categorized into few types. Some of the RDMA transports are either reliable or unreliable, and then, they are either connected or unconnected (also called datagram). With reliable transports, the NIC may use acknowledgments in order to guarantee in-order message delivery. Unreliable transports do not provide this guarantee.

However, modern RDMA devices for InfiniBand, Cray, or Omni-path implement a lossless link layer. The link layer prevents packet drops unless it is absolutely necessary, such as in the case of hardware failure or malformed packets. The bit based losses may occasionally require link layer re-transmissions. But, instead of dropping packets and re-transmitting like in Ethernet, the link layer will attempt adaptive congestion control and throttling to avoid congestion. Routing algorithms in link layer generally uses a credit based flow control to transmit and receive packets – i.e. a switch or host will only forward a packet to a destination if local credits are less than the destination *threshold* value. The same flow control mechanism is adapted for congestion control where credit thresholds are adjusted along all available routes according to the traffic state of the switches or hosts, effectively increasing or decreasing the flow. The link layer also tries to avoid deadlocks during network routing, thus user programs are never concerned with handling such scenarios. Cutting edge RDMA networks, therefore, can be considered highly efficient, as well as being reliable transports for data communication.

Optimization	**Description**
Transport switch	Select proper transport mode for different message lengths. Since MMIO is latency sensitive, it is more suited for small messages. The bandwidth friendly DMA transport mode(s) are generally more beneficial for larger messages.
Batch requests	If an application can issue multiple descriptors to a RDMA connection, then it can use one MMIO for the entire batch to initiate a DMA transaction for the entire batch.
Atomic Updates	Link level protocols may support atomic operations for RDMA. These may be useful for reducing extra round trip traffic for synchronous network operations where only few RDMA transactions need to be ordered.
Reduce signalling	Consider dropping certain DMA transactions, such as completion events if avoidable (i.e. polling). This decreases the number of PCIe transactions as well as NIC processing.
Descriptor shrinking	If payloads are in-lined in the descriptor (ie. small/medium messages), then any extra PCIe transactions may be reduced by in-lining all or part of the message in the header.
Header only/Inline request	Avoid the payload altogether if all or part of message can be embedded in the header or a completion event.

Table 2.2. **A List of Micro-Optimizations Possible with RDMA Fabrics.**

The user-mode NIC driver initiates RDMA operations by creating a descriptor (sometimes called a Work Queue Element (WQE)) in host memory; typically, these descriptors are pre-allocated in the contiguous memory region, and each descriptor may then be individually cache line–aligned. The descriptor format is vendor-specific and thus, is determined by the NIC hardware. Each descriptor may carry a payload along with some headers. The size of the descriptor depends on several factors, such as the type of RDMA operation, the transport, and whether or not the payload is referenced by a pointer field or in-lined [**71, 49**]. The descriptor itself may carry information regarding the type of network operation (RDMA PUT, GET, Atomic, etc) and the transport properties required (i.e. in-order/out of order delivery, completion signaling, etc). The size of headers varies significantly depending on both transport and operation and thus, have space for considerable micro-optimizations in the transport. A non exhaustive list of micro optimizations that are useful for optimizing RDMA software are listed in Table 2.2.

2.1.5. Local and Remote Completion. RDMA operations may carry some form of signalling event that indicates delivery notifications for data. NICs often use separate completion queues (CQ) to store completion event data and may also provide additional space to embed user data. CQs are a limited resource and, usually, the size of a CQ depends on the transport mode and NIC max queue limit supported by the hardware. The signalling events are important because they implement synchronization points for RDMA network transactions. RDMA buffers can be safely reused only if certain guarantees are made about the data delivery. Also, a delivery event may indicate that data was successfully received by the target node without any error or truncation to the payload. There are several types of completion events available in RDMA:

- **Local Completion** – events that notify the user on whether or not a local source buffer is available for future transactions. This may mean either that local buffer was copied to an intermediate buffer in the NIC and therefore, is safe for use, or it may mean that data from the source buffer was successfully transmitted to the target node. The later case provides the guarantee that NIC won't use the source buffer for any re-transmission, thus deeming the source buffer safe for reuse. However, in either case it is important to note that local completion may not imply that the data was delivered to the destination buffer.
- **Remote Completion** – events that notify the user if data is available for consumption in the destination buffer. Remote completion indicates a synchronization point for the target node. In this case, the event is usually delivered by the NIC to a remote queue. Sometimes remote completion can be emulated by polling destination buffers for a special tag or keyword. This technique, called "memory polling“, is often used to achieve synchronization.
- **Global Completion** – events that notify the user when entire network operation of copying source to the target was successfully completed. Unlike local events, these were triggered when data was successfully delivered from source to target memory. Furthermore, unlike local or remote events, these events will be made available through

both initiator and destination queues. A barrier is a common synchronization method utilized in global completion events.

2.2. RDMA Transport Libraries

The goal of RDMA transport libraries specifically, is to define a unified set of interfaces that enable a tight semantic map between applications and underlying fabric services. RDMA transports are designed for performance and for supporting as many platforms as possible. RDMA transports are also designed to be wrappers around native fabric interfaces such as verbs, PSM, PSM2, DMAPP or GNI. Due to the tight integration between network fabrics, the interfaces exposed are most often low-level and complex. Thus, they cannot be utilized directly in applications. The main audience for these transports are run time system and middle-ware developers. GASNet, UCX, libverbs and photon are some popular RDMA libraries in existence today. Figure 2.2 shows a common software design for a RDMA transport library. The lowest level of a transport is supported by the underlying operating system device driver. The native device drivers and support libraries are specific to each vendor and thus, vary by interface design. For example, Mellanox Infiniband devices have MLX4/5 device drivers and kernel level OFED verbs being supported by the operating system. In a similar vein, Cray Intel is comprised of their proprietary drivers and libraries such as Intel omni path and GNI device drivers and modules. It is common to find all public user-level interfaces defined on top of an operating system software stack. These generally correspond to lower-level APIs that implement RDMA memory domain registration, connection establishment, basic RDMA transport modes, and all relevant event handling for synchronization. Transport libraries introduce multiple software layers (typically with decreasing complexity) which build a unified set of interfaces for middle-ware and application consumption. While each layer may decrease the complexity of the RDMA interface to the user, it may also add additional overhead in terms of function calls, memory copies, and overheads in abstract data strutures, etc as we discussed in Chapter 1.

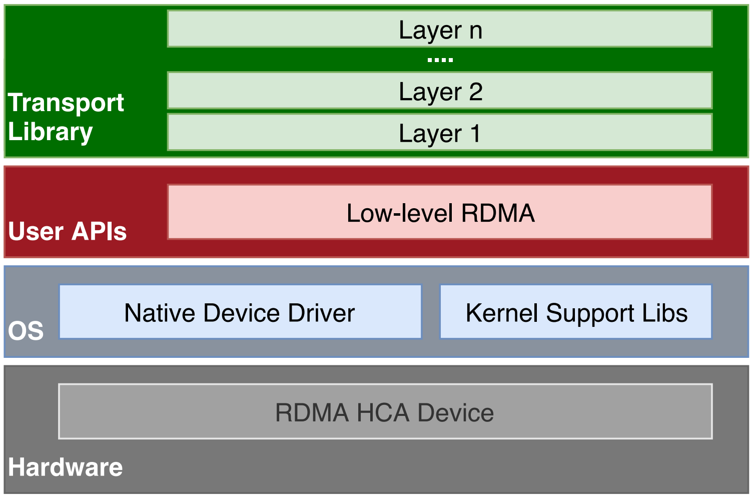


Figure 2.2. **A Common Architecture for a RDMA Transport**

2.2.1. Active Messaging with GASNet. GASNet started as an effort to provide a common HPC communication API for use as a compilation target by Partitioned Global Address Space (PGAS) parallel languages. These, notably, include UPC [**31**], Titanium [**34**], Co-Array Fortran [**88**], and OpenSHMEM [**29**]. Communication behavior in these higher level run time systems is characterized by one-sided, remote-memory-access (RMA) communication in distributed memory. The GASNet-1 API offers two primary modes of communication: (1) one-sided RMA interface and (2) Streamlined Active Message(AM) interface. The purpose of the former is to expose the RDMA capabilities of the network fabric enabling user-level RDMA enabled data structures for applications. Active messages enable a special case of RDMA primitives where communication is more dynamic and unstructured. An example of this is executing RMA operations on a dynamic graph on distributed memory. In this case, active messages will allow for processing vertices for newly discovered edges with RMA communication. GASNetEx, the latest incarnation of GASNet, has upgraded its capabilities to suit the exascale era. Some of the newer interfaces introduced include asynchronous event processing, offload APIs for supported NICs for expressing atomics, packing/unpacking collectives, and also, support for hybrid architectures, such GPGPUs.

2.2.1.1. GASNet RDMA Primitives. GASNet allows memory-to-memory and active message based RDMA between the host and the remote. GASNet describes its interfaces in terms of a core API and a network fabric independent extended API. The core API primarily contains functions for bootstrapping, job environment handling and active messages. GASNet bootstrapping is a two-step process. GASNet programs start by calling up `gasnet_init()`, which bootstraps distributed processes and establishes command-line arguments and the job environment. All processes then invoke the `gasnet_attach()` function to initialize the network and register shared memory segments. Active message communication (See Listing 2.1) is formed by logically matching the requests and replies from an initiator and a target process, respectively. Upon the receipt of a request, a special request handler is invoked. Similarly, once the destination responds a reply handler may be invoked at the initiator.

```
int gasnet_AMRequest*(gasnet_node_t dest, gasnet_handler_t handler,
                                              gasnet_handlerarg_t ...)
int gasnet_AMReply*( gasnet_token_t token, gasnet_handler_t handler,
                                              gasnet_handlerarg_t ...)
```

Listing 2.1. GASNet basic blocking extended API for RDMA

The extended API for GASNet (built around more generic low-level core API), entails the traditional one-sided model in which RDMA communication is decoupled from the synchronization. The blocking version of APIs (See Listing 2.2), wait until the operation is complete. These follow *global* completion semantics, and therefore, guarantees that the source data has been written to adhere to the destination memory. This also means that subsequent read operations issued by any thread (or load instructions issued by the destination node) will receive the new value or a subsequently written value.

```
void gasnet_get (void *dest, gasnet_node_t node, void *src, size_t nbytes)
void gasnet_put (gasnet_node_t node, void *dest, void *src, size_t nbytes)
```

Listing 2.2. GASNet basic blocking extended API for RDMA

```
gasnet_handle_t gasnet_get_nb (void *dest, gasnet_node_t node, void *src,
                                                           size_t nbytes)
gasnet_handle_t gasnet_put_nb (gasnet_node_t node, void *dest, void *src,
                                                                    size_t nbytes)
void gasnet_wait_syncnb (gasnet_handle_t handle)
int gasnet_try_syncnb (gasnet_handle_t handle)
```

Listing 2.3. GASNet basic non-blocking extended API for RDMA

Listing 2.3 reports basic GASNet APIs for non-blocking primitives. An invocation of these will return as soon as the operation is properly initiated. Each non-blocking function implements local completion semantics and therefore, once the initiation is complete, the source buffer will be safe for reuse. With `gasnet_wait_sync*()`, `gasnet_try_sync*()`, global completion semantics are implemented by periodically testing or waiting on the returned handle descriptor. There also exist other `gasnet_try_sync*_all()` and `gasnet_try_sync*_some()` variants, which will test for multiple completions for all or a subset of operations.

```
void gasnet_get_nbi (void *dest, gasnet_node_t node, void *src, size_t nbytes)
void gasnet_put_nbi (gasnet_node_t node, void *dest, void *src, size_t nbytes)
```

Listing 2.4. GASNet basic non-blocking extended API for RDMA with implicit transfers

```
void gasnet_wait_syncnbi_gets ()
void gasnet_wait_syncnbi_puts ()
void gasnet_wait_syncnbi_all ()
int gasnet_try_syncnbi_gets ()
int gasnet_try_syncnbi_puts ()
int gasnet_try_syncnbi_all ()
```

Listing 2.5. GASNet non-blocking synchronization for RDMA with implicit transfers

The implicit non-blocking API functions (See Listing 2.4) operate similarly to their explicit-handle counterparts, except they do not return a handle. They must as well be synchronized using the implicit handle synchronization operations (See Listing 2.5). These functions implicitly specify a set of non-blocking operations on which to synchronize. For example, there may be a set of outstanding, non-blocking, implicit-handle operations initiated by this process or thread (i.e. either all such gets, all puts, or all such puts and gets). The *wait* variants block until all operations in this implicit set have completed, while the *try* variants check the implicit set for completion [4]. Additionally, GASNet exposes convenient APIs for implicit *access regions* when split-phase synchronization is desirable. In split-phase, only a subset of operations may need to be synchronized immediately and then the rest can be fast-synchronized (i.e. in a batch) at a later phase of the program.

2.2.2. Multi-Tier RDMA Design – UCX. UCX is a communication framework with a two-level API design targeted at addressing the needs of next generation HPC systems. UCT, which is the lowest level interface, abstracts communication functions for various low-level networks. It adds minimal overhead to native RDMA interface, but also requires considerable effort from the user. The higher-level interface, named UCP, exposes a collection of high-level protocols, such as tag matching, RMA, and atomic operations, and simplifies the initialization of communication. A third component is also available called UCS: it provides the necessary services for memory management, data structures, and more. Currently, UCX supports many network fabrics such as InfiniBand Verbs [6], Cray uGNI [7], and shared-memory devices. Evidence has suggested [**91**] that UCP adds considerable overhead because of the extra software abstractions.

2.2.2.1. Communication Contexts. The high-level interface of UCX operates on a software abstraction called communication *context*. The UCP communication *context* structure abstracts RDMA memory regions for underlying devices. It is designed as a singular structure which dynamically selects a suitable interface for a communication operation, according to both hardware properties and performance criteria. UCT, on the other hand, abstracts devices and transports – the low-level interfaces capture both device memory and transport specific operations. For example, a UCT Infiniband UD interface may entail memory exchange, registration, and communication routines for Infiniband *Datagram* transport which

[4] `gasnet_try_syncnbi_*()` variants return `GASNET_OK` if operations have been successfully synchronized or `GASNET_ERR_NOT_READY` otherwise.

is connection-less. Importantly, a communication *context* structure may entail many UCT memory regions and thus, cater to specific user requests using appropriate device transport.

2.2.3. RDMA with UCX. UCT defines a set of functions for RDMA communication, categorized by remote one-sided, atomic operations and active messages. Thus, using these low-level function UCX implements different protocols for different message transfer methods. For example, RDMA put/get communication contains immediate (short), buffered copy (bcopy), and zero-copy message transfers. The short message protocols inline payload in RDMA headers with MMIO while *buffered copy* mode uses intermediate preregistered RDMA buffers for communication. The zero copy transfers normally involve larger messages and use the DMA capability of the hardware device. Thus, the implementation, in effect, depends on the hardware capabilities and device features in use. UCP API entails these low-level device specific functions and then offers higher level capabilities for RMA operations, remote atomic memory operations, and active message based tag matching. UCP also provides an internal multiplex between the appropriate transport protocol and performs message fragmentation, if necessary.

```
ucs_status_t ucp_put_nb(ucp_ep_h ep, const void *buffer, size_t length,
                    uint64_t remote_addr, ucp_rkey_h rkey, ucp_send_callback_t cb)
ucs_status_t ucp_put_nbi (ucp_ep_h ep, const void *buffer, size_t length,
                                        uint64_t remote_addr, ucp_rkey_h rkey)
ucs_status_t ucp_get_nb(ucp_ep_h ep, void *buffer, size_t length,
                    uint64_t remote_addr, ucp_rkey_h rkey, ucp_send_callback_t cb)
ucs_status_t ucp_get_nbi (ucp_ep_h ep, void *buffer, size_t length,
                                        uint64_t remote_addr, ucp_rkey_h rkey)
```

Listing 2.6. UCX High level UCP API for RDMA Communication

```
unsigned ucp_worker_progress(ucp_worker_h worker)
ucs_status_t ucp_worker_fence(ucp_worker_h worker)
ucs_status_ptr_t ucp_worker_flush_nb(ucp_worker_h worker, unsigned flags,
                                                    ucp_send_callback_t cb)
ucs_status_ptr_t ucp_ep_flush_nb(ucp_ep_h ep, unsigned flags,
                                        ucp_send_callback_t cb)
```

Listing 2.7. UCP API for RDMA Progress and Synchronization

Therefore, UCP exposes a single function for any message size. Moreover, UCP offers non-blocking communication functions for tag matching and RMA operations, which allow for the immediate reuse of communication resources, whereas UCT provides only non-blocking operations.

2.2.4. Designing for Built-in RDMA Completion Events. Photon is a customized transport layer that extends the traditional RDMA infrastructure provided by libraries such as GASNet. Its goal was to design a lightweight and efficient RDMA library for asynchronous communication on both rendezvous and one-sided models. However, it especially focuses on exposing a convenient abstraction for distributed memory global objects with a so-called messaging pattern in RDMA with completion events. RDMA with completion events sends variable-sized notification event data along with payloads to select and execute appropriate remote functions without any prior knowledge of the data (i.e. active messages).

As RDMA facilitates fast data movement without any cpu intervention, it decouples movement from synchronization and network progression. Synchronization support is necessary to implement advanced messaging patterns and high-level interactions of a messaging engine. CQ notifications supported by the RDMA fabric is mostly limited to a few bytes of data (i.e. immediate data). This native CQ support is not sufficient if users want to implement higher level remote procedure invocations and progress the state of complex distributed data structures. Thus, Photon tries to address this problem by providing not only a higher level, but also a high performance interface with variable sized notifications. It eases the burden on the user of handling the state of RDMA events by implementing appropriate tracking and control protocols for event data. Photon primarily introduces two types of RDMA primitives – Put With Completion (PWC) for RDMA write actions and Get With Completion(GWC) for read actions.

Photon PWC and GWC are supported by many network back end implementations for various RDMA technology such as Infiniband, Cray as well as shared memory(SMP) processors. As noted before, short completion event notifications are generally supported by the hardware abstraction (i.e. CQ). However, Photon strives to extend this facility to a first class RDMA primitive by exposing a generalized variable length notification event. To achieve this, Photon maintains a set of bounded buffers called "ledger". Each "ledger" may allocate variable length completion event data from already initialized RDMA buffers. Each "ledger" entry could be mapped to a remote or local identifier at the run time by a user RDMA call. The parameters and initial bootstrap for ledger configuration is provided upfront and completed before any PWC/GWC transaction. Initial bootstrap will also include an exchange of metadata for user buffers apart from ledgers.

```
// bootstrap
int photon_init(photonConfig cfg);
// buffer registration interface
int photon_register_buffer(void *buf, uint64_t size);
int photon_unregister_buffer(void *buf, uint64_t size);
// PWC/GWC RDMA and completion event handling
int photon_put_with_completion(int proc, uint64_t size, photonBuffer lbuf,
                    photonBuffer rbuf, photon_rid local, photon_rid remote, int flags);
int photon_get_with_completion(int proc, uint64_t size, photonBuffer lbuf,
                    photonBuffer rbuf, photon_rid local, photon_rid remote, int flags);
int photon_probe_completion(int proc, int *flag, int *remaining, photon_rid *request,
                    int *src, int flags);
```

Listing 2.8. Photon PWC/GWC API for RDMA Bootstrap and Communication function

Photon exposes both synchronous and asynchronous APIs (See Listing 2.8). A subset of these APIs are for buffer registration and exchange where all metadata for a particular RDMA action is marked as pinned and exchanged for secure communication. Photon users may utilize inbuilt info "ledgers" (initialized during Photon startup) for this purpose. Photon supports two types of RDMA operations, a) Rendezvous and b) One-sided. In Rendezvous protocol, internal Photon RDMA ledgers are utilized for send and receive like message exchange. For direct One-Sided, RDMA user space buffers have to be first registered and exchanged using relevant APIs described above and then, a PWC/GWC is initiated. Each photon back end uniquely handles RDMA requests according to a back end implementation (i.e. Verbs, PSM, GNI, etc) and also keeps track of them for bookkeeping. for example, when resources for ledgers are exhausted, requests are queued and progressed only when they are made available again.

PWC and GWC start by setting up RDMA operations with both payload and notification data from an initiator node to its peer. Each PWC or GWC action may first be configured to type and size of notification data (i.e. notification identifier). Each notification identifier is bound to a type of message or transaction that could invoke a particular action after a probe callback is executed. Then, a protocol is determined to efficiently deliver both data and payload. In Photon's eager protocol, RDMA payload is in-lined with the notifications. Thus, they are copied back to the user space RDMA buffers. The cost of an additional copy does not surpass the cost of two RDMA operations for small messages. However, for large messages, Photon uses the two RDMA operations since the cost of copying is larger. Thus, Photon initiates a direct RDMA fetch or put operation to remote buffers first, followed by another RDMA operation into the remote ledger for sending remote notifications. The ordering between these operations is important because the probe into notifications must always guarantee that data has already been delivered before consuming data. Photon utilizes internal completion notifications by HCA or NIC for RDMA on user space buffers and ledgers to enforce ordering between operations.

The abstraction exposed by Photon differs from GASNet or UCX such that it avoids low-level details of bootstrapping, completion event handling, and RDMA request tracking. Instead, Photon exposes RDMA with notifications as an extended first class RDMA primitive. Thus, the use of Photon may put less strain on application and run time system developers who hope to utilize native RDMA performance with a lesser code footprint. However, PWC or

GWC do not manage user-level RDMA buffers and this entire task is then left to the user to handle bootstrap and flow control protocols themselves. Although it is possible to re-purpose Photon's ledger based RDMA memory to a global heap, the current design makes it difficult to utilize it as a general purpose on-demand (i.e. loosely coupled) RDMA allocator within a distributed memory system.

2.3. Message Passing Standard

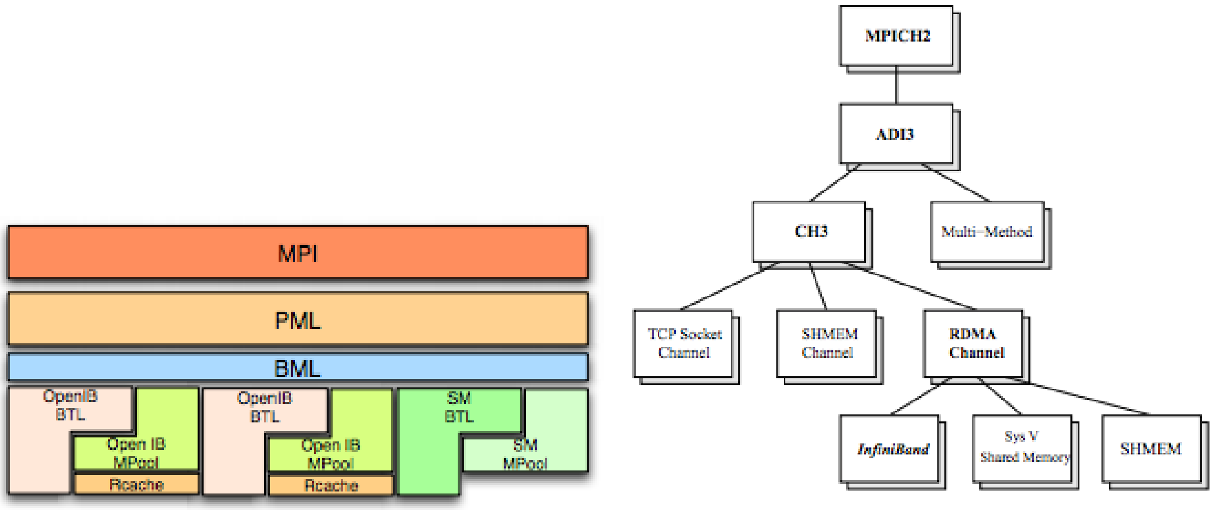
Message Passing Interface (MPI) is the de-facto standard [**54, 37**] programming model for peer to peer message passing in large scale parallel computing. It was originally designed for applications running on distributed memory architectures, such as clusters of single core machines and in most applications it is primarily used for this purpose. As multi and many-core compute nodes with shared memory hardware have become more popular, application developers have typically pursued a design where distributed memory MPI operations were used for inter-node communication. In contrast, intra-node communication was achieved by shared memory interfaces such as direct operating system support (i.e. IPC or POSIX shared memory), OpenMP [**27**], CUDA [**100**], OpenCL [**85**], etc, outside of MPI execution. The complexity of using multiple parallel programming models within the same applications makes it difficult to predict performance across run time and system boundaries. It is also important to develop ways for application developers to achieve high performance in shared and distributed memory architectures using a single abstraction.

MPI's distributed memory communication mechanisms within a single program multiple data (SPMD) mode of execution are a very good match for parallel applications. Specifically, MPI's distributed memory abstraction forces developers to clearly identify the information that is local to a core or remote, which encourages data locality. It has a near perfect interface for data parallel applications where compute operations are primarily performed on separate distributed memory domains independent to the communication interfaces provided by MPI, which separates the portions of the application devoted to computation and communication. However, the same design constraints have forced traditional MPI implementations to face performance constraints on modern hardware architectures. Some of these issues were highlighted by Chapter 1. However, it is important to look further into the basic MPI design, architecture, and programming model to disseminate some of these underlying issues.

2.4. MPI Runtime Architecture

Following the communicating sequential processes mode (CSP) [**70**], the core of MPI is based on communication between isolated processes. Each rank in MPI terms corresponds to an execution unit with its own address space. MPI defines two types of communication: MPI point-to-point is communication between two ranks. MPI collectives is communication between groups of ranks. In point-to-point mode, one process sends messages and the other process explicitly receives it. At the minimum, the process which sends data should declare the location of data, size, and the destination while the receiver should declare the location to read data and the source. MPI routines share optional tags which allow each rank to filter messages of certain type. This enforces additional constraints on the message passing implementation so that each send/receive pair is matched appropriately. In addition, MPI exposes a notion of a *communicator* which groups a selected number of processes into a single domain. This allows for the implementation of certain compute and communication phases to different groups of process either with or without overlap. This also means that now matching send/receive pairs should be a three-tuple with <*source/dest*, *communicator*, *tag*> combination.

2.4.1. MPI Communication Routines (Inter-node). Most MPI implementations carry multiple interfaces and abstraction layers for portability and reducing complexity for inter-node communication. A MPI implementation may contain many run time functions that assist the routines corresponding to message passing specifications [**37**]. These include process control, resource setup and tear down, network transmission management, file IO, latency awareness and management, fault tolerance, and optimized collective operations for common communication patterns [**59, 108, 115**]. Due to the fact that there is no standard for MPI run time, interfaces and abstractions have also become implementation specific.



(a) **OpenMPI MCA Component Architecture**

(b) **MPICH Channel Architecture**

Figure 2.3. **MPI Reference Designs** [**48, 52, 77**]

For example, MPICH[1] has a channel based architecture (i.e. AD3, CH3, etc) for portability and performance while OpenMPI has a component based modular architecture called MCA [**48**](See Figure 2.3). In general, run time functionality of an MPI implementation can be summarized by the following:

- Point-to-point Transport: This corresponds to a “wire protocol” of moving bytes between MPI processes. It is common to have these modules separate from MPI semantics. Multiple such modules can be used in a single process, allowing the use of multiple (potentially heterogeneous) networks, supporting RDMA (Infiniband, Ominipath, Cray uGNI, etc) or TCP/IP or even shared memory.
- Runtime Environment Support: For large scale applications, many MPI processes have to be spawned across clusters. MPI implementations may support these features by directly implementing them (i.e. remote spawn using a secure shell) or using underlying support of the cluster scheduler.(i.e. Slurm, PBS)

- Point-to-point Management Layer: provides message fragmentation, caching, scheduling, out-of-band messaging for base transport, and serialization and deserialization between an MPI layer and all transport modules.
- Collective Communication: the back end of high level group communication primitives in MPI collective operations, supporting both intra and inter-node communication.
- Parallel I/O: These may implement parallel file and device access on native and cluster based parallel file systems (i.e. Lustre, NFS)

One of the notable features in a MPI implementation is that it can take advantage of RDMA operations if they are provided by the underlying interconnect. These operations can be used not only to support MPI-2 one-sided communication, but also to implement normal MPI-1 communication.

2.4.2. MPI RDMA Communication Protocols. MPI defines four different communication modes: Standard, Synchronous, Buffered, and Ready. These four modes are implemented by exclusive or a combination of two internal protocols, Eager and Rendezvous. Further details on implementation choices for MPI and their relevance with RDMA characteristics and properties are described below.

MPI communication structures are initialized with the NIC/HCA both at the beginning of the run time (i.e. static initialization) and dynamically prior to a message exchange. In static initialization, MPI prepares message queues (or endpoints) and memory regions for regular messages for each logical node (i.e. ones that are already posted), as well as for messages which the engine does not recognize at present. These will generally end up in a separate *unexpected* message queue/endpoint. Additionally, MPI may also initialize endpoints for message acknowledgement for synchronous operations such as when an initiator requires a delivery guarantee on the target. However, in dynamic initialization, MPI may decide to initialize endpoints, memory, and matching descriptors, depending on the protocol running. An initialization phase generally must be followed by some form of barrier synchronization operation to guarantee that all RDMA related setup tasks are completed.

The initialization routine is highly dependent on the hardware structures supported by the RDMA vendor. For example, in Portals 3.0, low-level user space driver allows MPI operations to be indexed by a portal table. Similarly, Infiniband and Cray may configure CQs and endpoints that contain descriptors for pinned down RDMA memory regions. Indexing works by targeting remote process ID and a set of matched bits (i.e. 3 bits for protocol, 16 bits for communicator, and 13 bits for source rank). Thus, each communication operation may invoke a match table entry, CQ, or an endpoint that will contain the matching criteria used to complete the operation on relevant memory descriptors. Each memory descriptor may also be associated with an event queue signalling for completion.

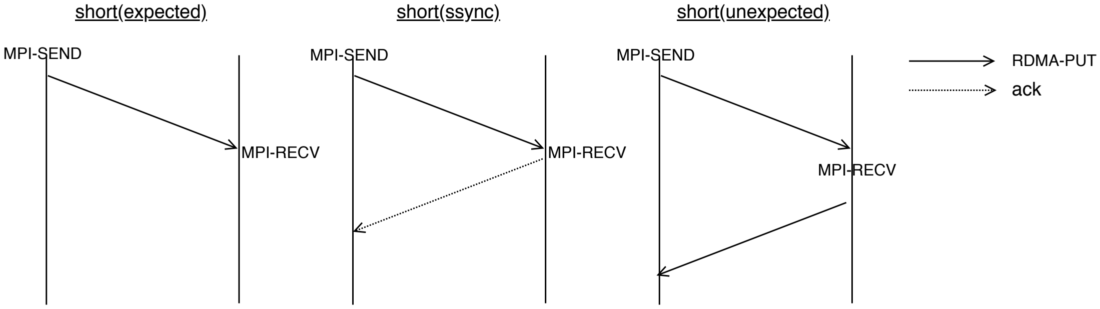


Figure 2.4. **MPI Short Message Protocol(s) with RDMA**

Communication protocols for short messages tend to follow an Eager pattern. In Eager protocol, a message is pushed to the receiver, regardless of its state. As mentioned above, registered RDMA buffer regions need to be pinned down before pushing messages to a receiver process. This protocol matches well with short messages because of the relatively low cost of copies and the design properties of underlying channel based send/receive RDMA. For example, both Infiniband and Cray support low latency send/receive RDMA channels for short messages with completion event notifications. MPI Eager mode sends a message along with other metadata, such as matching bits and headers in a single request. Unless matching receives are posted upfront, all such eager messages will be routed to an unexpected message queue. Before a short message is sent, the MPI library prepares remote and local memory descriptor on the initiator side with relevant matching headers. MPI may also configure event queues prior to a send request – this is important in detecting a delivery notification signal once a request has been successfully delivered (i.e. local completion event). Due to the fact that MPI specification requires matching send and receives to be posted before any message transfer takes place, eager protocols are only implemented in two specific cases. First, if the receive has been posted prior to the arrival of the send request, then the data is copied to the receiver buffer immediately. When an acknowledge response is sent back for `MPI_SSEND` synchronous requests to release resources at the initiator side and receives may be re-posted for control messages. For the latter case, if a matching receive has been delayed, then the message ends up in an unexpected queue until a corresponding receive is posted to move data from the temporary buffer. In such cases, the receiver side may initiate a second RDMA put request in order to comply with synchronous `MPI_SSEND` requests. MPI applications may pre-post `MPI_RECV`s to mitigate the cost of extra copies and synchronization. However, this strategy would not work for irregular and dynamic applications where a reasonable estimation of the upper limit for incoming requests is not always possible.

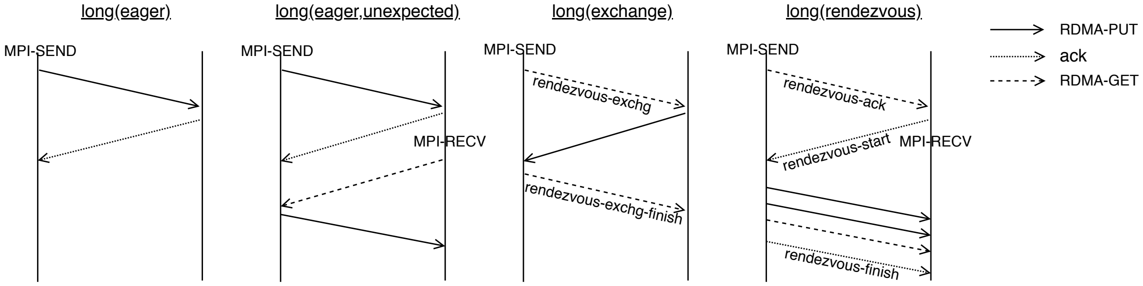


FIGURE 2.5. **MPI long message protocol(s) with RDMA**

MPI implementations may utilize the Eager protocol for long messages as well. However, since RDMA memory-memory operations require MPI library to know destination memory address beforehand [**79, 78**] [5], Eager protocols need to implement proper exchange protocols. Since copying is expensive for long messages and should be avoided, MPI Eager long protocol starts by bootstrapping metadata and caching them before any communication takes place. Pindown caching techniques [**107**] may be utilized to avoid bootstrapping for redundant buffers. In Eager long message protocol, MPI implementations assume that `MPI_RECV`s

[5] Channel based RDMA does not require destination address. However, Infiniband and Cray send and receive channel operations that only support small to medium messages.

have already been posted. This enables direct message transfer once matching is completed. However, when a match is not posted MPI has to resort to saving the matching state in an unexpected queue. In this case, a RDMA GET operation will be initiated when a corresponding matching is received. The state of completion for the `MPI_RECV/SEND` pair will be inferred by looking up event queue notifications. An acknowledgement notification with a successful match or RDMA GET completion event indicates completion of the operation. Some MPI implementations may also acknowledge that the initiator sent a RDMA PUT call instead of a passive RDMA GET operation. In either case, Eager long message protocol may waste link bandwidth and induce contention because delayed `MPI_RECV`s (i.e. unexpected) would lead to duplicate RDMA transfers.

To avoid the above issues, MPI implementations usually apply a rendezvous protocol for long messages. In Rendezvous protocol, MPI user buffers are pinned on-the-fly when `MPI_RECV/SEND` are executed. However, the buffer pinning and unpinning overhead could be reduced by using a pin-down cache technique. A reference rendezvous protocol always starts by bootstrapping buffer address data if not already found in its cache. The initiator will send a matching request but the data transfer would not take place until a matching post is received or acknowledgement is sent for protocol *start*. It is common to see either MPI initiator or receiver or both involved in actual transfer that utilizes the link bandwidth better. The protocol may also conclude with an acknowledgement on Rendezvous protocol *end* signal. Rendezvous protocol fits well with RDMA memory-memory operations, because they are powered by the bandwidth friendly bulk DMA transfer method. But the cost of buffer pin-down and registration coupled with message ordering guarantees may still hinder the performance benefits of Rendezvous transfer.

In addition to the cost of RDMA bootstrapping and exchange for MPI, other synchronization overheads are worth noting. A major source of synchronization in MPI comes from message matching. A typical MPI implements multiple levels of matching including wildcards. However, MPI considers matching on two message queues, namely unexpected and regular. MPI has to test messages in the unexpected queue first before posting receives on the regular queue because of the ordering guarantees of the specification [**37**] (i.e. No two MPI messages with the same communicator, tag, or source should be matched out of order). This check has to be executed atomically between testing unexpected queue and posting receives on a descriptor queue – in between these two states, no other message should be processed. Fortunately, hardware support for the atomic activation of descriptors is available. Descriptors may be enabled for pending events only after the unexpected queue has been tested for any matches. However, the protocol and overheads imposed by MPI matching is a major RDMA performance degradation factor.

RDMA fabrics generally feature both channel and memory semantics for MPI message passing. Channel RDMA operations are especially useful for Eager message mode, because Eager protocols do not require awareness of destination memory address. A channel message completion operation is implemented by polling CQs. While polling CQs is generally efficient with the hardware support available, latency for channel based RDMA tends to be larger than memory-memory RDMA because of the extra utilization of resources on the NIC. Furthermore, the MPI library could incur significant bookkeeping overhead for allocating and reallocating event descriptors on each CQ once the message is consumed. Instead, MPI implementations may opt for memory-memory RDMA for some of its messaging modes. One issue with this type of RDMA is how to enable notifications for messages.

Hardware support varies significantly such that some interconnect fabrics like Infiniband may not support CQ events for RDMA read/writes. In such cases, MPI may use two or more RDMA operations for data and notifications with necessary protocols for satisfying ordering between them. Traditional MPI RDMA implementations tried to solve this problem by polling on head and tail pointers by in-lining both data and notifications in a contiguous memory region. However, strict message ordering should be followed or supported by the hardware in order to achieve correctness in polling protocol. In the latest RDMA interconnects however, hardware support is available for synchronous RDMA notifications such as in uGNI where atomic or synchronous RDMA writes are made available to push notification flags in an arbitrary remote memory location once data is delivered. MPI implementations may utilize a hybrid approach for progressing the network by polling CQs as well memory polling [**78, 79, 59**]. In this case, the challenge is to find the optimal point of maintaining two polling sets without compromising overall performance.

It is also important to note that MPI specification enforces strict message ordering for message passing between two logical nodes. Thus, RDMA network adapters have evolved to feature messaging modes that support in-order delivery. However, with the invention of adaptive routing technology, cutting edge RDMA network adapters have showcased better throughput for out-of-order delivery modes and are often the default mode supported by the NIC/HCA. Unfortunately, this mismatch in MPI specification and hardware has resulted in various MPI implementations trying to work around the message ordering constraints via software protocols. For example, MPI implementations with packet sequencing implemented in transport layer have reported better performance [**80**] over traditional connection oriented RDMA protocols. While in-order delivery is useful for certain applications and messaging patterns, large portions of the application space may not require strict ordering, thus this can be considered a missed opportunity inhibiting a full utilization of the underlying hardware.

2.4.3. MPI3.0 RMA. The MPI-3.0 specification [**9**] defines a programming interface for exploiting RDMA networks directly. Extra copies and additional synchronization in MPI message passing protocols such as Eager and Rendezvous fail to leverage the full potential of RDMA hardware. In contrast, direct RDMA programming may alleviate some of these problems if the hardware is utilized correctly. In summary, the goals of direct RDMA programming are following:

- Attempt to avoid the costs of synchronization delays in programming model enforced barriers such as message matching and ordering, as well as minimize other side effects of bootstrapping, exchange, and synchronization in (unexpected) message queues.
- Attempt to gain memory savings by removing receiver side buffering for eager and similar protocols.
- Attempt to reduce energy wasted on moving data for message copies.

Accordingly, MPI-3.0 offers a large number of functions for direct RDMA with different performance characteristics for wide variety of use cases. Its RDMA functionality is categorized into three separate concepts: (1) window creation, (2) communication functions, and (3) synchronization functions. In addition, MPI-3.0 specifies two memory models. The weaker model, called "separate", defines a set of APIs that implement RDMA operations and synchronization on a generic programming model to support all types of RDMA hardware. It builds on the notion of public and private memory regions to support extreme instances where memory coherency for remote updates is not supported by RDMA hardware. Instead, it has to be implemented in software. A much stronger programming model, called "unified", is a subset of MPI3.0 RMA APIs utilized for highest performance. Here full hardware support is assumed and public and private regions are unified into one memory region, which is the case for all current RDMA networks.

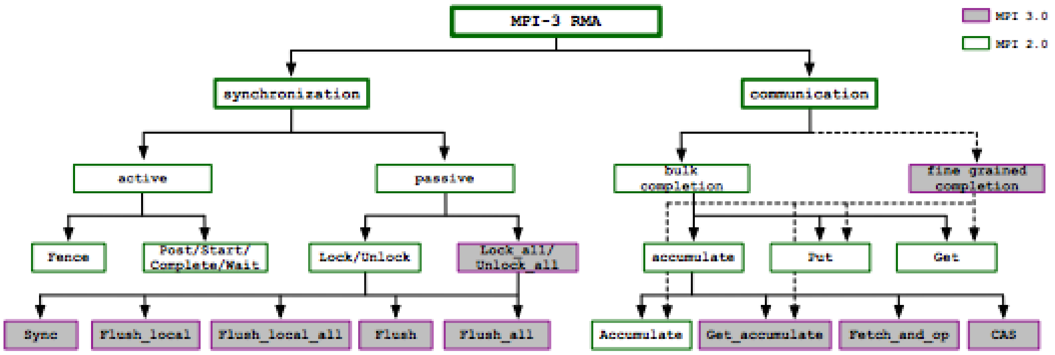


FIGURE 2.6. **MPI-3.0 RMA Communication and Synchronization Primitives** [9]

MPI-3.0 builds its RDMA abstraction around a primitive called a RMA "window". A "window" acts as the logical local or remote RDMA memory region whose processes can operate its RMA communication functions. RMA "windows" can be created either statically or dynamically during run time. The main complications for MPI-3.0 RMA programming stems from the separation of communication (remote accesses) and synchronization. Additionally, the MPI synchronization interface is split further into memory synchronization or consistency. For example, in memory synchronization, a remote process can observe a communicated value with a local RDMA read, while for process synchronization a remote process may gather knowledge about the state of a peer process. Furthermore, such synchronization can be blocking as well as non-blocking. MPI-3.0 communication interface exposes usual RDMA PUT, GET and number of remote atomic interfaces[6].

The MPI-3.0 communication interfaces are facilitated by both bulk (i.e. many RDMA requests at once) and fine-grained (i.e. one RDMA operation at a time) synchronization primitives for progression and completion. In active mode synchronization, all nodes are aware of synchronization operations. Active Interfaces such as `MPI_WIN_FENCE` are highly effective for synchronizing many RDMA PUT/GET operations at once globally. Alternatively, bulk synchronization may be implemented by split phase *active* operations such as `MPI_WIN_POST/START/COMPLETE/WAIT`. These so called access and exposure epochs allow memory to be exposed and closed to RDMA operations, depending on the current phase. Unlike global synchronization, split phase doesn't require all processes to participate in synchronization operation. Instead, subsets of nodes may synchronize RDMA communication operations, while others may work in compute operations. In passive mode only, the initiator invokes a synchronization operation. Passive Interfaces like `MPI_WIN_LOCK/UNLOCK/FLUSH` target a specific remote peer's memory region for updates and synchronization without the peer node being aware of the operation. Therefore, this type of MPI-3.0 RMA operation is more akin to native one-sided RDMA. An overview of MPI-3.0 primitives and architecture is illustrated in Figure 2.6 and explained in detail in MPI specification.

The consistency, completion, and synchronization of MPI models when treated as separate concepts allow the user to reason them out separately. RMA programming is, thus, slightly more complex [**111**] because of the complex interactions of operations. For example, MPI, like most RMA programming models, allows the programmer to start operations asynchronously and complete them (locally or remotely) later. This technique is necessary to hide single-message latency with multiple pipelined messages; however, it makes reasoning about program semantics much more complex. In the MPI RMA model, all communication

[6] Actual performance of atomic primitives is subjected to the amount of hardware support available

operations are non-blocking. In other words, the communication functions may return before the operation completes and thus, bulk synchronization functions are used to complete previously issued operations.

In the ideal case, this feature enables a programming model in which high latencies can be ignored and processes never "wait" for remote completion. The resulting complex programming environment is often not suitable for average programmers (i.e., domain scientists); rather, writers of high-level libraries can provide domain-specific extensions that hide most of the complexity. The MPI RMA interface enables expert programmers and implementers of domain-specific libraries and languages to extract the highest performance from a large number of computer architectures in a performance portable way.

2.5. Direct Memory Access on Shared Memory

While MPI has emerged as the de-facto standard [**54**] programming model for distributed memory architectures, it did not specifically cater well for shared memory architectures. This was because MPI was originally designed for applications running on distributed memory architectures, such as clusters of single core machines and in most applications, it is primarily used for this purpose. As multi and many-core compute nodes with shared memory hardware have become more popular, application developers have typically pursued hybrid designs where distributed memory MPI operations are used for inter-node communication and intra-node communication is performed via dedicated shared memory APIs such as OpenMP [**27**], CUDA [**100**], OpenCL [**85**], and the shared memory portions of the MPI specification. However, traditional implementations of MPI have faced performance constraints on modern shared memory hardware because MPI's run time was built on process-based isolation, which required unnecessary copying to communicate among MPI ranks on the same node. For architectures with a large number of cores (i.e. Xeon Phi [**30**], Knights Landing [**104**], TileraEx [**97, 12**], etc), the intra-node communication cost representing message copying became a significant setback. The limitations of process-oriented MPI implementations have motivated the research on implementations where MPI ranks are implemented as Operating system threads, all of which execute within the same process shared address space [**65, 92, 106**]. While threads give each rank its own stack and heap within the shared address space, the set of global variables are shared among all MPI ranks in a node. Thus, the application state becomes corrupted as different MPI ranks write to the common global variables, which may exist within the application and in any libraries that they link with. To mitigate the problem, cutting edge MPI solutions implement direct memory access routines between segmented parts of the address space for efficient intra-node communication. MPI Libraries (HybridMPI [**47, 46, 113**]) and Kernel extensions like XPMem [**120**], LIMIC [**69**], KNEM [**51**], Linux CMA (Cross Memory Attach) optimize communication by facilitating isolated address space sharing between processes coupled with fine-grain control over NUMA affinity for communication.

2.5.1. Processes for MPI Ranks. MPI specification does not prescribe how MPI ranks are implemented. Rather, the traditional assumption has been that each rank is an operating system process with its own private address space. The advantage of the process-based design is that it makes it easier to coordinate inter-node communication by multiple cores. Since each core is used by a separate process, their MPI libraries maintain separate state and thus, require no synchronization. Since network interfaces are typically designed to provide each process with a separate context in which to coordinate its incoming and outgoing communication, no run time synchronization is required by MPI to access the network.

The limitation of this design is evident in a scenario where ranks reside within the same shared memory node. Modern processor architectures share memory at multiple hierarchical levels (RAM, L3, L2 and L1 for example) for high performance. MPI libraries use first-in, first-out (FIFO) connections (one for each pair of local ranks) for small messages, and one or more shared memory region mapped by multiple ranks for larger messages. Thus, apart from the loss of cache locality, either case inherently requires two copy operations per message. The sender copies from its private memory send a buffer into the FIFO queue or shared memory region, and the receiver copies back out into its separate private-memory receive buffer. A common optimization for large messages is to overlap and pipeline the two copies by breaking the message into blocks, thereby allowing the sender and receiver to perform their respective copies simultaneously. FIFOs are a pair-wise connection, and shared memory regions may also be created on a pair-wise basis to simplify communication. Thus, the number of resources grows as does the square of the number of MPI ranks per node, which is often the number of cores per node. Such an approach will consume too many resources as the amount of memory available per core continues to decrease on HPC systems.

2.5.2. Threads for MPI Ranks. The limitations of process-oriented design has motivated research on implementations where MPI ranks are implemented as OS threads, all of which are executed within the same process [**46**]. Threads are a good choice because they share all their memory by default. However, many MPI applications are written with the assumption that global variables are private within each MPI rank. While threading gives each rank its own stack and heap within the shared address space, one set of global variables is shared among all MPI ranks in a node. The application state becomes corrupted when different MPI ranks are written to the common global variables, which may exist within the application and in any libraries they link with. Developers of thread-oriented MPI implementations have attempted to resolve this problem in two ways. First, they have developed techniques to privatize global variables, so that each thread is provided its own copy. At the source code level, privatization can be done using thread-local storage, using the **thread** keyword available in many C compilers, or by using compiler transformation tools [**86, 92**].

There has been work done on tools that modify object files to privatize global variables when the source code is not available [**86**]. Given the complexity of privatization, especially in library code, an alternative approach is to adjust the use of libraries to ensure that no globals are used. This involves replacing regular library calls with their thread-safe variants, for example, using strtok_r instead of strtok. Where thread-safe alternatives are not available (e.g., the getopt function uses static variables internally), locks are required to protect access to the function. While it is possible to build compiler tools to perform this replacement, they would require knowledge about each library and its thread safety guarantees.

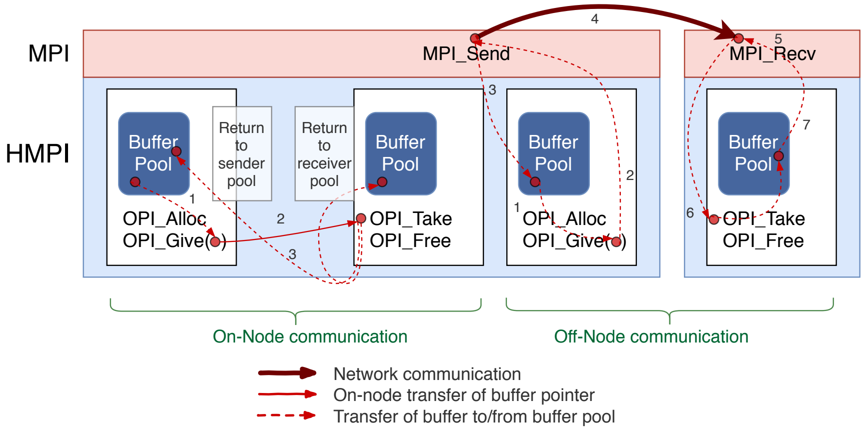


FIGURE 2.7. **Ownership passing method for fast intra-node communication between ranks by utilizing shared memory "Heap" allocators** [**47**]

2.5.3. Ownership Passing in Shared Memory. Friedley et al [**46, 47**], reported work to address the limitations of traditional MPI via a new MPI implementation called Hybrid MPI, which implements a heap that is shared among all the MPI processes that run on the same hardware shared memory system (e.g. a multi-core processor or a many-core device such as the Xeon Phi). The Hybrid MPI library addresses this by implementing a heap that is shared among all these processes, enabling them to communicate directly, as illustrated in Figure 2.7. A principle known as *ownership passing* [7] is utilized to enable MPI processes to communicate directly, thus fully leveraging the underlying shared memory hardware. The MPI library maintains per rank buffer pools which are utilized for safely acquiring or releasing memory segments for intra-node communication. Furthermore with *ownership passing*, efficient sender receiver pooling heuristics [**47**] have shown best results for practical parallel applications where messages (buffers) are often passed symmetrically between MPI ranks.

New many-core architectures, such as Intel Xeon Phi, Knights Landing [**30, 104**], offer applications a significantly higher power efficiency than conventional multi-core processors.

[7] *ownership passing* was primarily adopted from cache-coherency protocols [**68**] where implicit synchronized actions are leveraged for communication

Furthermore, they feature machine memory (i.e. RAM) which is shared among large number of cores that gets attached via product specific interconnect typologies. However, while this processor's compute and communication characteristics is an excellent match for MPI applications, leveraging its potential in practice has proven difficult because of the mismatch between the MPI distributed memory model and this processor's shared memory communication hardware. Thus, *ownership passing* technique seem to apply equally well for MPI applications executed on aforementioned architectures. In our initial implementation [**113**] on traditional multi-core Linux nodes, we used the mmap() Linux system call to map the same memory regions to multiple processes within the same OS instance. On the Xeon Phi, we pass shared memory configurations to Intel MPI using the environment variable I MPI FABRICS = <fabric >| <intra-node fabric>: <internodes-fabric>. Two kinds of shared memory (SHM) fabric modes are supported in Xeon Phi platform: a) "Local", which is used for communication among cores on the same Xeon Phi device and b) "SCIF-based" (symmetric communication interface), for communication to other devices. Intel MPI makes use of both to communicate amongst the disjoint address spaces of MPI processes. The system call mmap() is used to share data between processes on the same Xeon Phi device.

To enable data-sharing between cores on a Xeon Phi and cores on other processors, the SCIF-based mmap() is employed to map memory regions via a PCIe bus. This memory can then be accessed by any number of Xeon Phi coprocessors or any processors on the same node. Furthermore, the Xeon Phi CH3-SHM interface of the MPICH library, on which Intel MPI is based, implements the POSIX shared memory API for this memory. Data is communicated among different processes by copying the send buffer from the sender process' address space to an SHM buffer that is shared by the sender and receiver (Local for on-Phi communication and SCIF-based for inter-device communication) and then from there to the receive buffer. For small messages, memory regions are mapped as between process for each pair of cores and for large messages buffer pools/queues, which are maintained among processes regions. Although the exact implementation details are not publicly available, references on Intel MPI library suggest that it employs similar pairwise regions for SHM fabric mode.

Large message pools employ pipe-lined message copy for improved performance. Unfortunately, because this design requires message data to be copied through an intermediate buffer, it is inherently less efficient than the direct communication provided by shared memory programming models. Our Xeon Phi port of Hybrid MPI uses `mmap()` of Xeon Phi SHM fabric, with some modification to malloc() and related functions to share the entire heap of all the processes running on Xeon Phi cores. The key idea of Hybrid MPI, illustrated in Figure 2.7, is that a large shared memory buffer is allocated on a given device and each MPI process on that device is assigned an equal non-overlapping region within this buffer to utilize *ownership passing* techniques. In the Busy Box OS, this is achieved via a predefined file on an SHM device (`/dev/shm`) within the tmpfs pageable/swappable file system. Due to the fact that all MPI processes have access to the entire shared memory segment, they can directly access each others' heaps. We have also utilized 'MAP SHARED' to enable transparent page updates across a shared region, along with the 'MAP FIXED' `mmap()` flag to assign a fixed virtual address space for all the ranks. Although it is theoretically plausible for a rank to have a shared memory chunk that exceeds the available physical memory by using pageable swap memory on disk, the limitations of the Intel Xeon Phi/KNL design make this nearly impossible [**30, 104**]. The Xeon Phi has no directly accessible disk drives by default and for this reason, its root file system is stored in RAM disk (tmpfs). This limits the amount of available RAM and swap pages cannot be supported unless appropriate drivers/modules are installed for networked file systems such as NFS.

2.5.3.1. Intranode Communication Protocols. Above implementation uses *ownership passing* to implement MPI point-to-point communication operations more efficiently than the native Intel MPI implementation. However, in the current Hybrid MPI implementation, collective operations such as MPI_Bcast are not implemented in this manner and rely solely on the underlying Intel MPI implementation. The message passing techniques we discuss in the next sections are based on the zero-copy operations performed within the shared memory region provided by Hybrid MPI (message passing literature does not count the copy from the send to receive buffer, so the Hybrid MPI design is zero-copy, while Intel MPI is single-copy).

MPI point-to-point communication requires two steps. First, an arriving messages is matched to an MPI_Recv operation on the receiving process. Then, the message data is transmitted. For each MPI process, some number of FIFO communication channels can exist, each identified by the rank of the sending/receiving process, an MPI communicator object, and an integer tag. Each MPI_Send and MPI_Recv operation specifies the channel it uses and a channel descriptor (rank, communicator, tag tuple) is then attached to each message. The tuple on each arriving message is used to match it to an appropriate receive operation posted by the receiving MPI process. For each shared memory domain (set of processes connected by hardware shared memory) Hybrid MPI allocates a single queue into which all messages communicated among processes in the same domain are placed. Note that this is only possible because the address space of all processes is shared and thus, they may all access the same memory locations. MCS locks [**83**] are used to manage concurrent access to the queue in a cache-conscious manner.

2.5.3.2. Direct Transfer. Since all processes on the Xeon Phi share the same heap, by default, messages are transferred directly from the sender to the receiver buffer using a memcpy() operation, without any additional copies. Unlike traditional single-copy mechanisms where the sender copies a message into a shared buffer and the receiver copies it back out, in Hybrid MPI the transfer can be performed by either process, since both have direct access to both buffers. In Hybrid MPI, the receiver performs the direct transfer, since it is the process involved in message matching in this case. Thus, immediately after a sent message is matched, its data is copied to the receiver's buffer by the receiving core. There can be situations where either the receiver or sender's buffer is not in the shared heap segment allocated by Hybrid MPI. For example, the application may use other memory allocators (e.g. POSIX memalign()) or the text, or stack segment to store application buffers. In this case, Hybrid MPI performs an additional memcpy() to transfer data.

2.5.3.3. Immediate Message Transfer. MPI communication requires messages to be both matched and communicated. As such, in the immediate protocol, Hybrid MPI merges message matching and communication by copying the content of the MPI message immediately after its header. The result is that the communication required for the matching (the header) causes the hardware prefetcher to start transferring the message data without incurring an additional cache miss. Hybrid MPI uses this protocol for messages up to 512 bytes in size. Message buffers are aligned to start at cache line boundaries using memalign().

A many core architecture's memory subsystem plays a crucial role in this protocol. For example in Xeon Phi, each core is equipped with an 8 way set associative 32 KB of L1 private data and instruction cache and 512KB shared inclusive (for each point in time, L2 cache has a copy of L1 cache) cache. If the sender and receiver continue to stride on the same memory region (i.e. the sent buffer is contiguous in memory), then the L2 streaming hardware prefetcher enables Hybrid MPI to achieve very low communication latency and high throughput. Experiments on this have indicated that Hybrid MPI improves bandwidth and latency for small sized messages relative to Intel MPI for Xeon-Phi [**113**]. The intra-node communication time to transfer messages between 32 bytes and 8KB, showed that Hybrid MPI outperforms Intel MPI, with 65% lower communication time corresponding to 6GB/s peak bandwidth for the buffer size range when the immediate protocol is used (over 32b - 512b messages) and 43% and 70% higher bandwidth.

2.5.3.4. Synergistic Message Transfer. Although the direct protocol makes it possible for either the sender or the receiver cores to transfer the message, performance is further improved by involving both cores. Thus, for messages larger than 12KB, *ownership passing* employs pipeline transfers by breaking the message into smaller blocks and allowing both sender and receiver to engage simultaneously whenever possible. Here, the receiver initializes the protocol by transferring the first 12KB block and then, incrementing a shared counter. As more blocks are transferred, the processes used the counter to identify the next block to transfer until the entire message has been communicated. Messages between 8KB and 12KB are transferred in two blocks synergistically by the sender and receiver. In the sepcific case of the Xeon Phi architecture, a bidirectional ring interconnect was present, which deemed extremely useful for synergistic data transfer. Due to the bidirectional nature of the ring interconnect and physically distributed nature of memory controllers, every core on the ring was able to communicate with each other on an average shorter number of hops (compared to a bus-like interface), making this type of data transfer very efficient. Thus, at the hardware level, core pairs can utilize least cost channels to transfer data within the interconnect when engaged in synergistic transfer [**113**].

Furthermore, all cache levels follow a fully coherent MESI-based [**68**] GOALS coherence protocol, making any prefetched cache available to other core during synergistic transfer almost immediately. For example, if the receiver has prefetched one or more cache lines that stride over to the next block that belongs to the sender (to be transferred during its synergistic time), then the sender would not incur any additional memory access or TLB costs. By pipelining the transfer using synergistic protocol, *ownership passing* enables Xeon Phi's bidirectional interconnect to efficiently interface with both cache and GDDR5 via memory controllers, making the maximum use of the memory channels. For example, during the memcpy() phase of synergistic transfer, multiple pairs of ranks can use multiple channels available at the ring simultaneously for the load/store operations. With each channel having the ability to extract 5.5 GT/s with an access frequency of over a 32 bit wide memory bus, Xeon Phi was reported to produce up to 350GB/s theoretical aggregate bandwidth via the memory channels.

In the inter-node case, rank pairs always reside on two different physical nodes, which causes all message transfers to be performed via remote RDMA read/write operations over the network. In practice, the utility of the synergistic protocol will depend on the application's communication patterns, including the overlap between the sender and receiver's communication operations and the overall utilization of the interconnect. Intra-node communication bandwidth for MPI with *ownership passing* technique reported a peak bandwidth of 50GB/s in Xeon Phi which was significantly higher than the peak 40GB/s achieved by native Intel MPI. More importantly, the insight gained from *ownership passing* principle paved way for improving inter-node communication with the core work described in this thesis. Our task was also greatly complemented by the advancements in RDMA technology that features lossless link level protocols [**121, 53, 71**], fine grained event support and synchronized (atomic) updates [**13, 50, 73**].

2.6. Direct Memory Access on Distributed Memory

The focus of this work is to explore techniques and methods to leverage RDMA for Distributed memory. The *ownership passing* principle described above is a generic memory allocation algorithm implemented only for shared memory allocation pools. However, thanks to the advancements in RDMA, this principal may be adopted with modifications for distributed memory. Shared memory synchronization performance for segregated memory allocators [**75**] have been thoroughly studied and documented [**74, 41**]. Going by the performance challenges in shared memory, the primary challenge for a distributed memory allocator would be to explore novel ways to minimize synchronization cost to facilitate peer to peer communication. Apart from various work on statically partitioned global address spaces (PGAS [**29, 57, 36**]), dynamic allocation for distributed memory using segregated allocation techniques have not been explored. This thesis is the first work of its kind to study design and implementation of a distributed dynamic memory allocator with RDMA.

Newer RDMA abstractions such as RSockets [**62**], Remote Regions [**6**], Demikernel [**122**], TCP/IP-based RMA [**82**], Channel-based heterogeneous RMA [**94, 110**] and DPDK [**25**] have shown relevance towards data center applications and have started to leverage capabilities of RDMA. Aguilera, et al's work called Remote Regions [**6**] reports, how a simpler interface than existing low level RDMA (i.e. Verbs, libfabric, PSM, uGNI or GASNet/UCX) may be utilized to export process local memory to other nodes. However their work is based on exposing direct memory access capability via Linux file system abstraction and thus require modifications to OS kernel to utilize in applications.

CHAPTER 3

Towards a High Performance RDMA Transport

"Perhaps I could best describe my experience of doing science in terms of entering a dark mansion. You go into the first room and it's dark, completely dark. You stumble around, bumping into the furniture. Gradually, you learn where each piece of furniture is. And finally, after six months or so, you find the light switch and turn it on. Suddenly, it's all illuminated and you can see exactly where you were. Then you enter the next dark room."
— Andrew Wiles, (1953-Present)

Natively one-sided higher level programming models such as PGAS and MPI 3.0 RMA utilize low-level network RDMA libraries such as GASNet [**11, 61**], Gravel [**33**], ARMCI [**87**], GASPI [**57, 58**], UCX [**101**], and Photon [**73**], for remote memory access and zero-copy data transfer. These software offer more control over RDMA network operations and provide notification events that facilitate RDMA operations by indexing into interconnect specific data events on completion queues [**71**](CQs). For example, a user application may direct the network adapter's RDMA engine to perform either a RDMA PUT operation, which instructs the engine to move data from local to remote memory, or an RDMA GET operation, which directs the engine to move data from remote to local memory and to then notify a source and destination upon completion. The type of event varies by hardware even for different HCAs of the same vendor [1]. These general notification events only indicate that the data transfer has been completed or that the local buffer is available to reuse.

[1] For example, `GNI_CQMODE_LOCAL_EVENT` is only available for Cray large message transactions, which cause an event to be delivered to the local endpoint's CQ when the local DMA engine (*a.k.a.* BTE) has finished handling the descriptor data

3.1. Motivation

The synchronization overheads incurred by high level parallel runtimes such as MPI [**96, 98, 108, 50**], is a major obstacle for achieving the highest possible performance from the underlying RDMA networks. While user-space RDMA libraries may provide near native performance, they are not parallel application friendly because of the lowest level of abstraction provided. The development and maintenance of direct RDMA programming in application or middleware could easily become a nightmare due to the complexity involved in implementing efficient flow control and communication protocols. Thus, this thesis is driven by the need to search the middle ground for high performance RDMA substrates that will co-exist with both existing run time systems, as well as expose sufficient abstraction that manages subtle differences in performance characteristics of the hardware to extract maximum performance. The motivation behind such RDMA transport layer is 4-fold. They are listed below:

- *Time* – Gain savings in response time and latency by minimizing unnecessary or wasteful synchronization, such as global barriers, bootstrapping, or in Rendezvous (i.e. software message re-ordering and matching) and optimizing network layers with efficient synchronization protocols for different transport modes.
- *Space* – RDMA memory could be fragmented across the cluster. Thus, methods should be developed to utilize RDMA memory efficiently. Furthermore, extra copies may waste valuable network and processor resources and therefore, should be avoided in all possible instances.
- *Energy* – Reduce energy spent on extra data movement, signalling requests, and acknowledgements by leveraging hardware support.
- *Abstraction* – Expose convenient or familiar abstractions for direct RDMA programming, as well as avoid overly restrictive constraints that may inhibit true RDMA network performance.

Notification mechanisms would need to be implemented by transport libraries, considering the specifics of each network interface adapter and its performance properties of data transfer modes, completion delivery, and signalling. In most cases, notification event data are made available on a connection basis. Cray uGNI provides few APIs to modify CQ event data. Some fabrics (like Infiniband/Mellanox) support per message event data in headers. While such mechanisms are useful as synchronization events, these methods are not scalable as they are a limited resource. A more generalized event notification system can be realized for remote completion with hybrid polling techniques [**117**]. Polling consumes fewer resources on the network adapter or the Network System on Chip (SoC), because it does not need to manage CQs (tracking interrupts, completion entries, manage overrun events, etc.) and/or interrupt mechanisms to notify the receiver. Instead, control is delegated to the receiver itself. In terms of ordering guarantees, polling fits nicely with fine-grained notification events because out-of-order events would not necessarily match the polling order by user.

While CQ or memory polling is useful, it needs to be implemented judiciously. Polling consumes extra space on receiver side buffers, therefore, careful consideration needs to be assessed on the memory footprint and bandwidth required to communicate polling flags. Also, network transaction initiators must ensure that payload data has been transferred to the remote end before the polling headers – not satisfying this requirement would result in the applications receiving stale or invalid data. Another consideration would be whether or not to implement buffer “in-use”/“re-use” synchronization semantics in a network substrate, a property that cannot be guaranteed solely by completion/notification events. Many network libraries [**11, 101, 55, 3**] and low-level APIs have left the responsibility for synchronization to high-level run times. We observe that handshaking, credit-based flow control, and coarser synchronization (i.e. MPI-3.0 Active RMA) protocols must be utilized with caution, so that they do not undermine the benefits of RMA in distributed memory applications.

Since middle-ware libraries work more closely with the fabric and may provide maximum utilization of the network adapter/hardware, their target audience has always been runtime system developers, who will, in turn, use them as tools to build higher level communication primitives and programming paradigms. One of the overarching issues of RDMA is the efficient handling of the transport layer. For instance, neither local completion nor remote completion semantics alone will indicate if a RDMA buffer is safe for a network operation again. More specifically, a remote RDMA buffer transaction is in flight. We call this buffer "in-use"/"reuse" semantics [**117**]. Thus, transport protocols should be designed to manage safety guarantees and correctness, while not overly compromising on performance.

3.2. High Performance RDMA

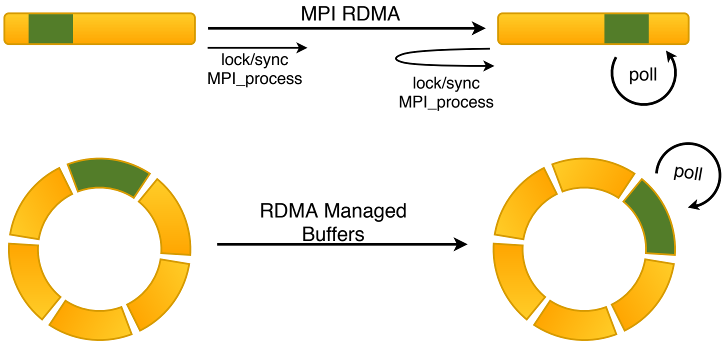


FIGURE 3.1. **MPI vs rmalloc Transport**

In order to solve the problem of appropriately managing RDMA, runtime systems implement sophisticated flow control protocols such as credit flow control, counting barriers, or bulk synchronization [**121, 13, 9**] (i.e. MPI-3.0). Although bridging the gap between application and the underlying network fabric via a high-performance, low-level middleware interface is highly desirable, from a programmer's perspective, building such sophistication directly into an application is a major hindrance. Furthermore, commonly used synchronization protocols such as credit flow control and persistent circular buffers may not be an optimal space efficient solution. For example, in cases where RDMA access is irregular and RDMA-enabled memory regions are externally fragmented, applications may stall unnecessarily until corresponding remote buffers are made ready for particular RMA operations, concerning the "reuse" state may not be enabled for it until all or a subset of "in-use" buffer regions transition into the "reuse" state, effectively under-utilizing RMA-enabled address space available.

We posit that RDMA could be better leveraged by providing a low-level, but simplified (i.e. familiar UNIX-like interface) programming model, which fully utilizes a given interconnect for accelerating RDMA operations by eliminating overheads of intermediate layers and enabling scalable synchronization. **RDMA Managed Buffers** [**119**] is a novel RDMA transport that better leverages state-of-the-art RDMA support while presenting an interface that exposes internally managed RDMA enabled memory for peers. Our transport library (*aka* RMALLOC), manages RDMA memory by producing RDMA capable memory segments through a distributed RDMA memory allocator instance [**118, 117**]. It is important to note that it updates the local channel state when memory is consumed, but remote endpoints are notified only on-demand, as more memory is required. This approach minimizes synchronization overhead that otherwise occurs when global barriers are used or when a RDMA window is expanded. Due to the fact that buffers are managed internally, our event-driven model allows applications to utilize completion events and ameliorate network back-pressure to avoid stalls by lightweight optimized network progression. RDMA Managed Buffers also help to increase throughput of application network exchanges, by overlapping communication with computation. Thanks to the simplified abstractions for zero-copy messaging and RDMA events, RDMA managed buffers act as a transport-level communication substrate accessible to both applications and run time systems equally.

The main components of RMALLOC library and **RDMA Managed Buffers** programming model are listed below:

- An event-driven RDMA transport library for message passing. This exports transport management functions for remote RDMA memory allocation and event/polling based synchronization. It also implements a bootstrap management layer that handles the initialization of RDMA Managed Buffers instances on different environments and run time systems.
- A distributed dynamic memory allocator called “rmalloc”, which allows for efficient utilization of RDMA-enabled virtual memory segments. To achieve this, “rmalloc” employs multiple heuristics, including: next-fit, best-fit, and worst-fit for allocation

operations. It also uses a protocol named bootstrapping channels to accomplish high-throughput RDMA allocations between endpoints

- A lazily initialized Unix pipe-like specialized RDMA communication construct called "rpipe" which binds with the remote allocator. It exposes remote data access operations on distributed reader/writer processes. It also enables effective communication patterns for RDMA Managed Buffers transport that are applicable to parallel applications and kernels, such as zero-copy point-to-point messaging, halo exchange, active messages, and sparse graphs

3.3. Design Considerations

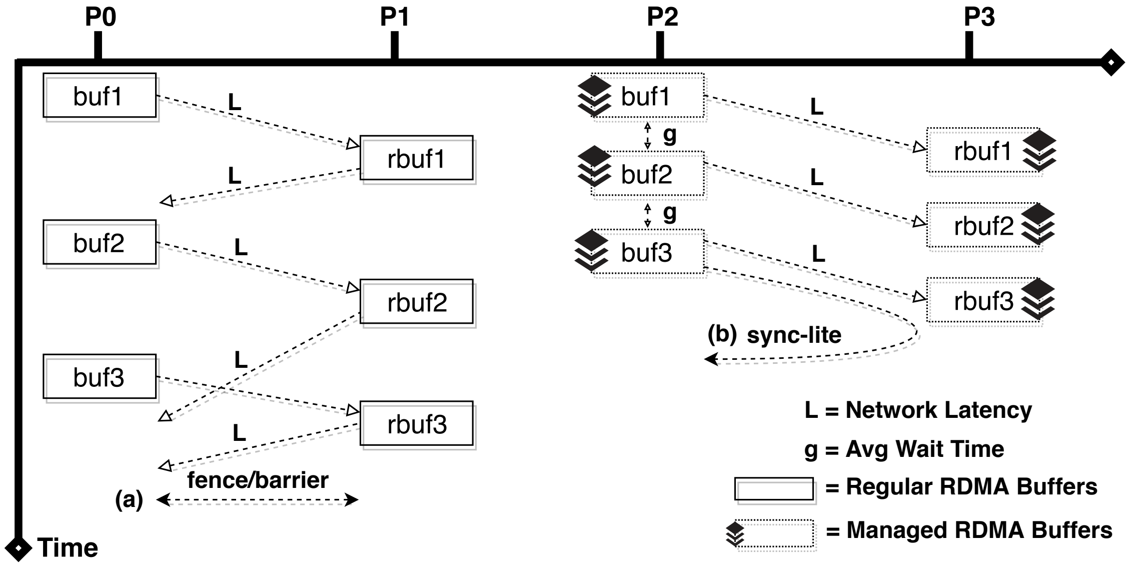


FIGURE 3.2. **Coarse-grained barriers vs. fine-grained event synchronization**

Effective utilization of RDMA in software systems involves considerations on the nature of the communication, as well as handling interconnect-specific details as efficiently as possible. For inherently two-sided communication, RDMA performance gains are inhibited by model-imposed barriers. For example, in MPI Rendezvous Phases called "pinning" and "exhange", matching constraints enforced afterwards incur a cost for each communication operation.

MPI-2.2/3.0 tries to reduce these costs by utilizing remote memory windows [**37**] and ameliorating synchronization overheads by enabling access/exposure epochs. For cases where RDMA reads and writes are irregular, coarse synchronization may become unfavourable, owing to a lack of fine-grained control that negatively impacts applications by reducing their ability to overlap communication and computation. Enforcing synchronization is often confusing and complicated for passive targets (i.e. `MPI_Win_lock` and `MPI_Win_unlock` and `Win_flush`) – the causality model of local and remote completion events may be masked by the specification details [**123, 9**] (i.e. supported consistency model) and thus, the application's performance could depend entirely on the MPI implementation and platform support available.

Coarse-grained synchronization barriers, fences, and counters are widely utilized for synchronizing access to RDMA buffers despite the possibility that they may lead to global stalls or under-utilization of some of the compute resources. Alternatively, fine-grained control over synchronization [**13, 50, 73, 61**] is desirable for efficient RDMA as it avoids global stalls in the system and thus, better utilizes resources. Completion events are the primary building blocks around which flow control synchronization is built in RDMA network protocols. They declare the points at which RDMA buffers are ready for reuse in local or remote endpoints. To sufficiently describe synchronization safety for RDMA operations, in addition to the information delivered by completion events, further exchanges are required to gather information about the communication state. MPI one-sided communication routines use synchronization-heavy barriers or epochs [**9**] to satisfy this requirement. Figure 3.2 highlights two generic scenarios where RDMA buffers are synchronized differently for one-sided communication. In the first case (a), two processes (i.e. P_0 and P_1) exchange messages with three RDMA buffers by bulk synchronization. Here messages are consumed only after P_0 and P_1 have synchronized by waiting for completion events for all processes to reach completion. Examples are MPI 3.0's epoch based `MPI_Win_fence` or PSCW [**9**] phase barriers – messages are exchanged by synchronizing RDMA transfers in bulk. Despite the obvious drawbacks, bulk synchronization of communication has the advantage that no software state needs to be maintained to identify discrete network operations.

Some parallel programs require a more fine-grained control of local buffer resources and thus, must be able to complete specific messages on demand. This sort of fine-grained RDMA communication is defined by MPI2.0/3.0 *locks* (i.e. `MPI_Put, MPI_Get, MPI_Win_lock/unlock`) and *request-based* RMA programming (i.e. `MPI_Rput, MPI_Rget`), although a significant run time cost is generally associated with managing request contexts and flow control synchronization through a heavyweight messaging engine. Figure 3.2(*b*) shows two processes that utilize RDMA buffers which are internally, but lightly managed and synchronized (*aka* sync-lite) by an intermediate software layer (i.e. RDMA *Managed Buffers*). Here the software does not employ explicit request management, message ordering, matching, or expensive global or flow synchronization. Instead, a RDMA memory allocator intercepts all allocation and deallocation requests and then notifies a software managed channel (i.e. established between the peers) when the peer state need to be updated. Since buffers are managed internally, both P_2 and P_3 do not have to wait for completion events and may even be able to withstand network congestion delays by returning control to the user. Therefore, P_2 and P_3 inject messages at a much higher rater than P_0, P_1. Since the next available RDMA buffers are determined by the library, if P_2, P_3 do not wish to wait for all endpoints for completion, then RDMA *Managed Buffers* may opt to complete synchronization with whichever buffers are available for communication.

3.4. Architecture

Figure 3.3 shows the design of RDMA *Managed Buffers* transport. RDMA managed channel layer transparently establishes a link between local and remote instances and provides low-level public RDMA routines, `rmalloc`, `rread`, `rwrite`, etc. It also facilitates bindings with the dynamic memory allocator (aka RMALLOC), which prepares RDMA ready virtual memory for underlying network functions for network operations ahead of time (similar to

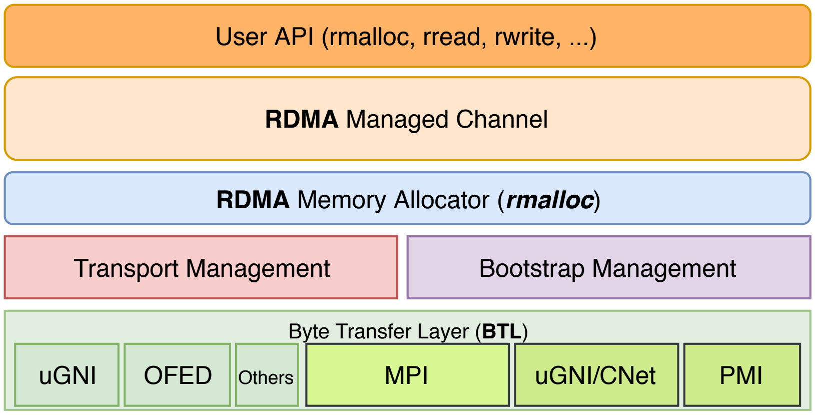

User API (rmalloc, rread, rwrite, ...)
RDMA Managed Channel
RDMA Memory Allocator (rmalloc)
Transport Management
Bootstrap Management
Byte Transfer Layer (BTL)
uGNI
OFED
Others
MPI
uGNI/CNet
PMI

FIGURE 3.3. **Design of rmalloc RDMA transport library**

MPI-3.0 RMA Windows), but will be provisioned on-demand using a dedicated communication channel between a source and destination process.

3.4.1. Application Layer. The application layer presents user facing interfaces for channel configuration, creation, memory management, and RDMA operations. Primary RDMA operations "rwrite" and "rread" are exposed as automatically synchronized interfaces. Primary consideration with this strategy is that programs should not be concerned with synchronizing remote memory accesses. For example, when a process is executing a RDMA write to a remote memory region two types of conflict could arise. First, the remote memory may not have been consumed by the corresponding remote process thus, running the risk of overriding data. Second, any in-flight requests on the same memory region may still be in operation, therefore, this may invalidate partial or full updates. Instead, "rwrite" and "rread" may perform the necessary synchronization operations and any checks for data consistency will be performed in the background to ensure consistent updates. The public API also contains operations for configuration and creation of RDMA memory channels (a.k.a. RDMA "rpipe") between two peers .The configuration of a RDMA channel prepares it for different RDMA put or fetch type operations, sizes, and granularity of RDMA allocation or type of allocation heuristics that are supported (such as next-fit or best-fit). Additionally, configuration properties may contain network fabric specific parameters such as a maximum number of completion events per channel, low level transport mode for different message sizes, and thresholds for fine tuning. Application layer features "rmalloc" and "rfree" as main memory management interfaces, which transparently provisions memory on remote nodes via an already established RDMA channel ("rpipe"). The memory management and channel interfaces are inspired by Unix and C Standard library functions (i.e. `pipe, malloc, free`), which are commonly used for inter-process communication (IPC), albeit ours is an extended version of distributed processes (i.e. readers and writers) that maps a local and remote endpoint. This presents an excellent opportunity to expose a simplified programming abstraction for RMA, while still maintaining sufficient performance/throughput for communication operations. Apart from public APIs, the application layer also facilitates lower level library APIs for more fine-grained control of memory management, synchronicity, (i.e. asynchronous and synchronous) and network operations.

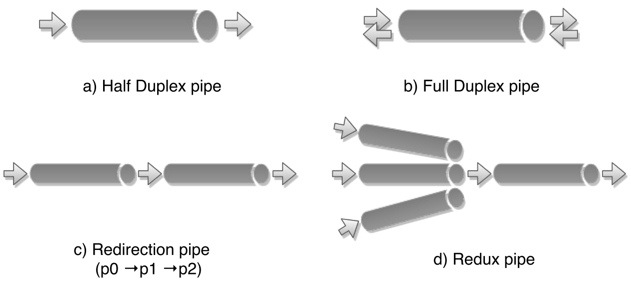


Figure 3.4. **Construction of rmalloc channels**

3.4.2. RDMA Managed Channel. A managed RDMA channel ("rpipe") encapsulates all RDMA memory allocators. Depending on the type of operation supported (RDMA Put/Get), a "rpipe" is established with an appropriate RDMA allocator configuration suitable for RDMA communication between two peers. A "rpipe" will, at minimum, host two allocators - a *local* and a *remote*. A *local* allocator typically contains RDMA enabled heap memory required for processing the local portion of a RDMA operation. A *remote* allocator is a shadow memory portion that mirrors RDMA memory in a peer. For example, RDMA Put will use a local allocator for source operand and remote allocator for writing to memory on the destination. The shadow allocator consistently and reliably provisions requested memory from the free remote memory regions before executing RDMA operations on the remote peer. A primary requirement for a channel to synchronize free RDMA memory in a remote peer without compromising on a RDMA operation's performance is the existence of a shadow allocator or heap. However, this task is complicated by the kind of memory allocation heuristic that is supported. A next fit heuristic will always select the next available slot in a contiguous memory region and thus, require a different synchronization protocol than the more generalized allocation heuristics, such as best-fit or worst-fit. Synchronization protocols may be implemented either in channel or network semantics. In network semantics, a channel will make use of low-level network capabilities for protocol setup, while in channel semantics, any number of RDMA channel types will be utilized. Therefore, RDMA channels may contain these additional channels called "bootstrap" that satisfy requirements for channel and network synchronized protocols for different heuristics. The most basic "rpipe" RDMA managed is a "half duplex" RDMA channel between two endpoints where RMA transfer happens only in one direction. Alternatively, a "full-duplex" channel may allow RMA transfers in both directions. Other complex pipes such as *"redux"* and *"scatter"* may also be constructed by aggregating primary pipe constructs.

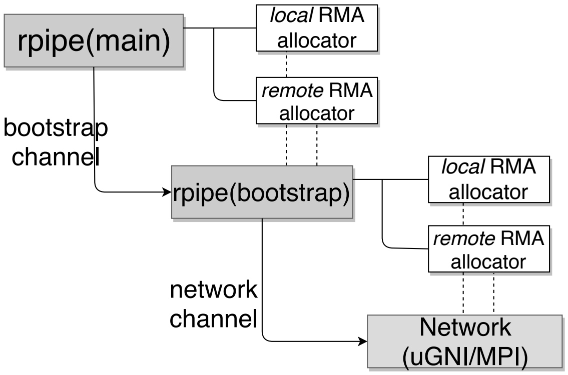


FIGURE 3.5. **An Architectural view of a rmalloc channel**

3.4.3. Remote Dynamic Memory Allocator (rmalloc). Unlike in MPI, *rmalloc* completely eliminates the exchange of remote memory addresses during network operations; once the dynamic RMA memory allocator instance has been set up, it will always return an RMA-enabled memory address for an RDMA operation. Synchronization semantics for safety and correctness are designed to be implicitly contained in `rmalloc()` and `rfree()`

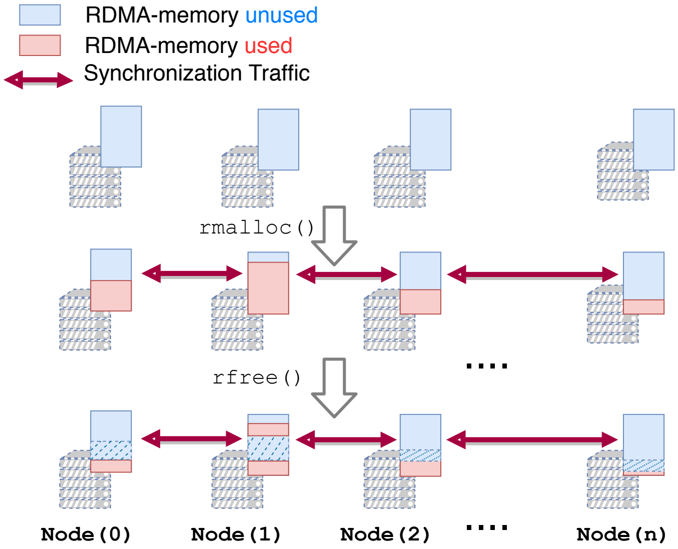
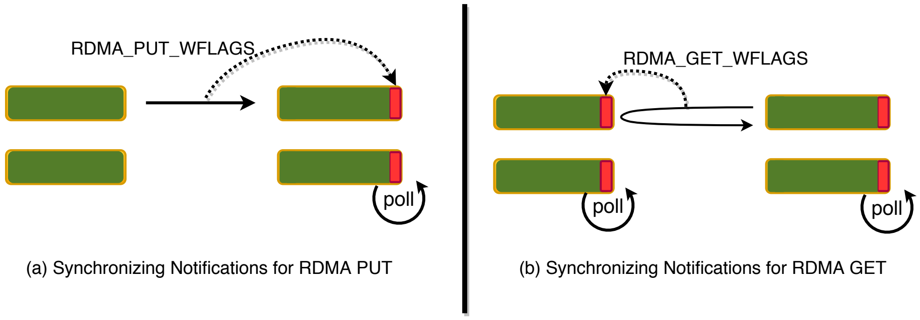


FIGURE 3.6. **rmalloc RDMA heap regions in a distributed memory system.**

operators. Therefore, our model stipulates that `rmalloc()` memory is in "reuse" state, but transitions into "in-use" state immediately after the successful return of a network `rwrite()` or `rread()` operation. The RMALLOC allocator is similar to other dynamic memory allocators like `malloc` *and friends*. However, it is also able to provision a contiguous range of RDMA *ready* virtual memory on both local and remote nodes for the purpose of data access and synchronization. The RMALLOC heap memory regions are made RDMA *ready* by two ways. First, it registers these memory regions with the network adapter. This step is essential, since remote DMA requires physically pinned memory to correctly operate. How exactly this is achieved depends on both the network fabric and the facilities available via the user space driver software. Second, RMALLOC handles different alignment and constraints required for RDMA memory regions as needed by the currently active network transport mode. For example, certain transactions in Cray may require 32 byte alignment for synchronized header data, while other types of fabrics or transport modes may not impose such restrictions. Thus, RMALLOC is responsible for managing such situations transparently while exposing RDMA enabled memory to the applications. Most importantly, remote allocation (*aka* shadow allocation) serves a special significance in distributed memory allocation. It provisions memory in support of a (physically) remote process while keeping in sync with the remote state[2]. Figure 3.6 presents a snapshot of memory layouts for RDMA heap instances in a distributed memory system. As shown, the system starts with all heap instances set up with finite memory regions available for RDMA operations. With each node allocating memory (i.e. via RMALLOC ()) for communication of a parallel program, the memory regions may become fragmented with time. Some of the acquired regions may also be released over time as buffers get consumed. These state transitions require some form of synchronization exchange in order to maintain consistency for local and remote (shadow) allocators. Thus, synchronization protocols for achieving this play a critical role in handling on-demand RDMA memory for a performance efficient operation.



(a) Synchronizing Notifications for RDMA PUT

(b) Synchronizing Notifications for RDMA GET

FIGURE 3.7. **rmalloc RDMA Heap Regions in a Distributed Memory System.**

3.4.4. Transport and Bootstrap Management. Granularity of notifications of data delivery and receipt for RDMA operations is a highly important consideration when designing RDMA transport protocols. Additionally, the software should be able to handle data movement efficiently enough to control traffic to maintain notifications and so that the transport state is minimized. The transport management layer is tasked with handling all message exchange and synchronization protocols and fine-grained notifications. As displayed by Figure 3.7, for RDMA PUT operations, *atomic* updates (i.e. synchronized updates) are made to the message headers as soon as the RDMA operation is completed (`RDMA_PUT_WFLAGS`).

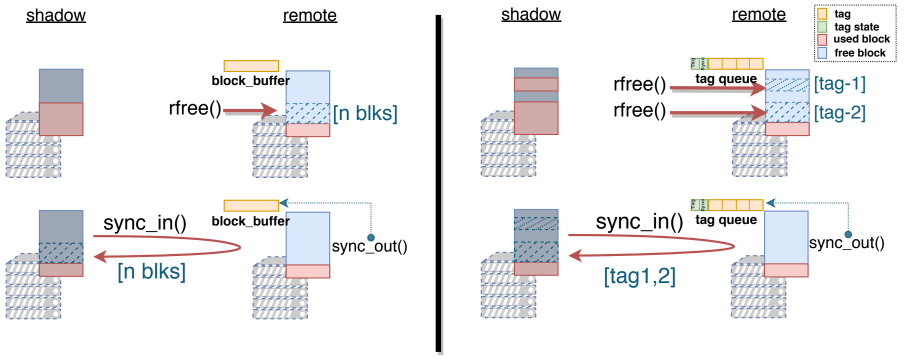
[2] Thus, the allocator can be considered a distributed process

Atomic update mechanism allows RDMA memory to be polled at the passive [3] remote side to detect a notification of data receipt. Meanwhile, an active initiator never has to wait for completion as long as the same memory location is not accessed again. A pull based synchronization operation (i.e. RDMA GET) is performed with the peer if the same memory location needs to be accessed again. Such synchronization operations are named `sync_in`. They will ensure that the data delivered was successfully consumed by the remote process before any memory region utilized for prior RDMA operation is returned again by the allocator. Similarly, for RDMA GET, atomic updates are made to the source headers in memory as soon as RDMA operations are completed. In this case, however, completion at the source active side may be detected using a flag updated by a synchronous RDMA fetch operation `RDMA_GET_WFLAGS`. This approach to fine-grained synchronization is a hybrid scheme called "notified flags with polling" – we mix both polling and CQ events to implement notifications[**117**]. Each RMALLOC allocation generates a memory offset for special tags useful for polling. RDMA operations notify the endpoint of completion by writing directly to a tagged memory, which is offset once the underlying fabric has guaranteed that data movement is complete (via local CQ events). It also exploits the fact that although RDMA-enabled memory is a limited resource, the space required for polling tags comes relatively cheap. Furthermore, this method scales well with allocation size and consumes fewer resources on the network adapter or the System on Chip(SoC), because it does not need to manage CQs which track completion entries, manage overrun events, and/or handle interrupt mechanisms to steer the receiver, instead, control is delegated to the receiver itself. In terms of ordering guarantees, "notified flags with polling" fits well with the fine-grained notification approach, because out-of-order events are handled directly on the address space returned by RMALLOC . While various vendor devices support a wide variety of hardware features for fine-grained notification events, each feature comes with different levels of constraints based on the network platform architecture[4]. Furthermore, the type of synchronization event varies by hardware support available on the device [**71**]. Thus, the

[3] passive, because in pure RDMA, remote process is not involved in the data transfer

[4] For instance, Cray network adapters impose memory alignment requirements for synchronization flags for Fast Memory Access (FMA) PUT transactions in Aries, which is not the case for the Gemini.

lowest level of byte transfer "back end" framework (*aka* BTL) abstracts platform specific RDMA details for various networks such as uGNI(Cray), libverbs(Infiniband/Mellanox), Infinipath PSM(Intel), and others, and then implements fast-path RDMA base operations for the transport. The overall design strives to provide a simple synchronization free "front-end" for fine-grained, lightweight, event-based RDMA programming and an extensible framework for supporting native performance across the board.



(a) Synchronizing shadow state by blocks

(b) Synchronizing shadow state by allocator tags

FIGURE 3.8. **The rmalloc deferred synchronization modes for RDMA shadow and remote memory allocator instances.**

The synchronizing state of a shadow (local) and remote instance is necessary to consistently provision RDMA memory regions between peers, as well as reuse memory used for RDMA operations. One strategy for executing this operation is to send synchronization packets with each outbound or inbound RDMA network operation. This protocol may be implemented by establishing a bootstrap RDMA channel (i.e. A RDMA channel exclusive for synchronization) in the direction of shadow-to-remote instance. A bootstrap channel will write each allocation descriptor tag onto one end of the pipe, while the remote node will be able to read each tag to update its state. However, this method is network latency sensitive, because a remote node has to wait for RDMA write completion of each tag before memory could be provisioned on the instance. Therefore, a block synchronization protocol is frequently utilized to synchronize the state between remote and shadow allocator instances. For this protocol, any number of contiguous memory blocks are selected if they are no longer being used. But, unlike channel based protocol, it does not send sync packets for each allocation request. Instead, block synchronization will fetch (i.e. `sync_in`) a number of blocks from a remote block buffer that were freed by the remote process. This may happen once shadow memory capacity reaches some minimum threshold or full. This allows for allocator instance states to be synchronized sparsely with one-sided operations, without incurring heavy synchronization traffic.

One limitation with this method is that blocks are always in contiguous chunks of memory. The aforementioned operation is only best suited for next-fit heuristics, because it allows for acquiring and releasing memory contiguously from the last block. Other heuristic algorithms such as best-fit, worst-fit, or random-fit won't always acquire or release memory contiguously. A more efficient protocol is sometimes needed for heuristics that deal with non-contiguous allocations. To alleviate this problem, synchronization may be implemented by exchanging several bytes-long tag descriptors stored in RDMA enabled bounded queues. Each tag may correspond to a rmalloc allocation request with information containing the virtual memory block start and end boundaries and also, other metadata such as network header information. Each queue may carry state data such as tag state which indicates if queue updates are valid and also synchronization state – whether queue is ready to use again once it became full. In the shadow and remote instance, a context holds information about exchange state. Typically, two tag contexts will be implemented in order to track allocated tags and free lists. Each context is associated with a RDMA queue containing tags that are outbound (`sync_out`) and inbound (`sync_in`) by synchronization traffic. When a memory descriptor is released, it is appended to the outbound tag queue, which will be asynchronously fetched by the remote peer when synchronization is required as reported by Figure 3.8(b). Similarly, new allocations are communicated to the remote peer by fetching from outbound RDMA tag queue. Synchronization modes and its relationship to allocation heuristics are summarized in Table 3.1.

Synchronization Mode	Heuristic Supported	Properties
Bootstrap Channel	Next-fit, Best-fit, Worst-fit, Random-fit	1. High latency 2. Immediate Synchronization 3. No Contexts
Network Block	Next-fit	1. Low Latency 2. Deferred Synchronization 3. Single connection context for accumulating blocks
Network Tag	Next-fit, Best-fit, Worst-fit, Random-fit	1. Low/Medium Latency 2. Deferred Synchronization 3. Multiple connection contexts for allocated and freed up tags

Table 3.1. **A list of synchronization modes available with rmalloc.**

The bootstrapping management layer handles all common network tasks that are required to be able to initialize RDMA channels. These include: buffer management and key exchange routines, message passing, and coarse synchronization methods, such as network wide barriers. This functionality is essential because most high-performance network fabrics require metadata exchange using a separate out-of-band network, such as TCP/IP, prior to making them RDMA ready. RMALLOC transport may implement bootstrap functionality in either one or two levels, depending on the underlying network back end. In one-level implementation, all bootstrap is achieved by a single layer. In two-level implementation, RDMA bootstrap is achieved by a out-of-band network, while delegating control messages to an fast bootstrap network on the native fabric. Furthermore, metadata exchange may be delegated to a higher-level run time system, such as MPI or a low-level interface, such as the process management interface (PMI [**8**] or PMIx [**24**]), which generally take advantage of a cluster-wide process launcher, such as Slurm or PBS. Additionally, this layer exports useful locality information via the underlying bootstrap networks or process management interfaces.

3.4.5. Network Byte Transfer Layer (BTL). Architecturally, the lowest layer of the “rmalloc” and “rpipe” library is the network “back end”. This layer operates directly with a network adapter's hardware abstraction layer or an SMP node. The library exposes key networking functionality in "back end" such as RDMA_PUT, RDMA_GET, and atomic operations for the overall operation of "rmalloc" and "rpipe". Additional functions, such as the creation of secure network domains, the creation of completion queues, the registration of virtual memory, the initialization of logical endpoints, and data transfer are also handled by a "back end". Among currently supported "backends" are uGNI, MPI, Infiniband for inter-node and SMP for shared memory. The "rpipe" communication layer loosely integrates with a RMA memory allocator and network back end. A bootstrap network is a special type of network "back end" that is made available through BTL. Essentially, these interfaces are utilized as services by the upper-level bootstrap management layer for RDMA channel bootstrapping. As noted before, our library strives to avoid coarse-grained synchronization and therefore, implements both completion events and polling via network APIs. This hybrid scheme, also called "notified flags with polling" , works as a basis for local and remote completion semantics.

CHAPTER 4

Implementation

"The greatest strategy is doomed if it's implemented badly."
— Bernhard Riemann, (1826-1866)

This chapter describes the reference implementation details of RDMA *Managed Buffers* transport. Main implementation aspects can be categorized in two ways. First, we must take a look at transport protocol implementation for bootstrap and remote allocation functions. I consider them the highlight of this thesis as they play an integral part of building a high-performance RDMA network software. Secondly, we will look at how network functions are built from the ground up for the types of network fabric discussed. These low-level tasks facilitate the creation of secure network domains/completion queues, registration of virtual memory buffers, initialization of logical endpoints, actual network RDMA operations, and building up synchronization operations with fine-grained RDMA events, including network hardware assisted functions when possible.

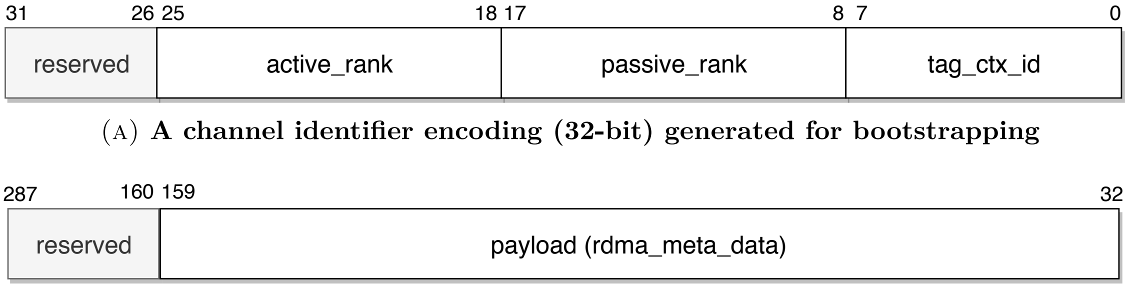
4.1. Bootstrapping

As explained in chapter 2, unlike TCP/IP, RDMA enabled, high-performance network fabrics can not be made ready for communication the first time. Instead, messaging modes on the fabric must be setup properly before any communication can take place. Metadata exchange and endpoint establishment are, therefore, a critical first step for utilizing RDMA. Many environment specific methods may be explored for bootstrapping. For clusters with process launch environments built in, such as Slurm or PBS, PMI could be utilized for the task. Using higher level run time systems such as MPI is another alternative. However, in this case, transport implementations should be careful to not conflict with existing RDMA channels of MPI system. Finally, transport implementation could make use of complete out-of-band networks, such as TCP/IP over Gigabit ethernet. We explain the general bootstrapping mechanism of the reference implementation and how control networks are implemented as utility service to achieve this task in the following:

Algorithm 1 BOOTSTRAP_EXCH_NB(P_B, XSTORE, GET_EID, $ch\_id, peer, mdx, size$)

```
Input: Bootstrap network $P_B$, Exchange store XSTORE, GET_EID returns a network specific unique id for an exchange request, $ch\_id$ is the unique RDMA channel id of the caller, $peer$ corresponds to the process with which RDMA exchange takes place; if this process is the RDMA active side, then it will initiate a send operation, $mdx$ points to the bootstrap data being exchanged of length $size$ bytes.
Local: Let $Hb$, $Hr$ denote hash table buckets corresponding to RMALLOC instance unique identifiers $ch\_id, r\_id$ which were requested and received respectively. Let $tmp$ be a temporary buffer of length $size$ bytes.
if ISACTIVE($P_b$)
    ret ← SEND($P_B$, PEER, $mdx$, SIZE)
else
    ret ← RECV($P_B$, PEER, $tmp$, SIZE)
    $r\_id$ ← GET_EID($tmp$)
    $Hr$ ← XSTORE[GET_HASH($r\_id$)]
    $Hb$ ← XSTORE[GET_HASH($ch\_id$)]
    $b$ ← SEARCH_BUCKET(H$b$, $ch\_id$)
    if $b \equiv$ NULL then // No match found for current request
        if $ch\_id \equiv r\_id$ then
            COPY($mdx$, $tmp$, SIZE)
        else
            HB_INSERT(H$b$, $mdx$)
            MATCH_RECV_QUEUE(XSTORE, GET_EID, $r\_id$)
    else
        COPY($mdx$, $b$, SIZE)
        MATCH_RECV_QUEUE(XSTORE, GET_EID, $r\_id$)
```

4.1.1. Initializing Network Back Ends. Each network "back end" is initialized by a synchronized call to a `rinit()` which triggers the execution of a *bootstrapping* network. A bootstrapping network handles all message routines required for common network management, buffer, and key exchange routines. For example, an MPI bootstrap network initializes the MPI "back end", which creates MPI-3.0 RMA windows with `MPI_Win_create _dynamic()` and `MPI_Win_attach()` routines. A *bootstrapping* network has two stages for a uGNI-based "back end". An MPI or PMI bootstrapping code exchanges network information in order to initialize a special GNI-based control message network. If the first stage bootstrapper is MPI, then the connection domain identifiers need to be remapped in order to avoid conflicts with MPI. Note that MPI might have been booted earlier on the same network driver with a predefined set of connections, which may lead to connection conflicts. Once initialized, a message queue-based control network protocol takes over to prepare the GNI "back end" for endpoint creation and binding routines (i.e. `GNI_CqCreate`, `GNI_EpCreate`, `GNI_EpBind`). The GNI bootstrap is built on uGNI FMA Short Messaging (SMSG)[1] facility that enables users to send short messages between endpoints on separate nodes.



(A) **A channel identifier encoding (32-bit) generated for bootstrapping**

(B) **Meta-data layout for 128 byte payload**

FIGURE 4.1. **Meta-data exchange packet layout for *bootstrapping***

4.1.2. Bootstrapping Channels. RDMA *Managed Buffers* implements a control message (CM) network to facilitate the exchange of metadata (secure domains, keys, RDMA buffer locations, etc) among endpoints. Once initialized, the CM network takes over to prepare the network "back end" for endpoint creation and binding routines. CM Network is responsible for the creation and lazy initialization of a RDMA channel object. Algorithm 1 refers to the protocol implementation for the aforementioned bootstrap exchange between nodes. `bootstrap_exch_nb` is a generic non-blocking service that is made available to all network "back end" implementations. The interface for this service is designed such that any incoming requests for RDMA channel creation is handled asynchronously without any awareness for network "back end" implementations. For example, `bootstrap_exch_nb` is

[1] A SMSG message is a special form of FMA PUT transaction using a messaging type sometimes referred to as a "mailbox".

not concerned with a uGNI specific metadata handling structure. Instead, the routine will copy these data back to the network layer for further processing, if the request was successful or otherwise if action is deferred to a later time. MDH exchange store is a hash-table that tracks unbounded number of unmatched channel creation requests. Each request is indexed by a unique key. If any of the items match current channel keys, then the metadata is forwarded to the network and the item is removed from the hash bucket. If no matching item was found, then the received key will be matched against the input network channel key. Upon successful matching, exchange data will be forwarded to the network as before. However, in the unlikely scenario that received and input keys are non-matching, keys are pushed back to the exchange store for future processing. As reported by Algorithm 1, the exchange makes use of two important communication functions (i.e. send, recv) of the bootstrap network. Two assumptions are made by these bootstrap network routines. The send interface is assumed to be non-blocking and follows local completion semantics, while a receive may wait for completion.

For an exchange to complete, each channel instance will be assigned a unique key by a combination of source and destination nodes to store its bootstrap data in a table. Figure 4.1 and Figure 4.1a represents the overall layout for bootstrap transport packet and the initial region comprising channel identifier encoding for it. The layout stores active and passive rank number in bits 8 to 25. Given two logical nodes A, B, this coding allows for distinct identification of a RDMA channel established between $A \longrightarrow B$ from $B \longrightarrow A$. The context identification number (i.e. `tag_ctx_id`) encodes each RDMA channel context. It enables a bootstrapping maximum of 256 (i.e. 2^8) RDMA channels between a pair of ranks. The second region of the bootstrap packet (Figure 4.1b) contains the actual metadata stored as a contiguous buffer region. The format and size of the metadata region depends on the network fabric driver which is initialized during library setup phase. While the above is the default channel tagging mechanism, the library also allows different custom encoding, if underlying network fabric provides native channel tags such as UUIDs available in infinipath [**35, 14**] PSM.

4.1.3. Allocation Policies.

The *rmalloc* implementation supports well-known allocation policies and has extensions to do the same for others in the future. However, for the synchronization stage, we will only discuss next-fit policy for the remote allocator instance of *rpipe*. Synchronization for other policies is still at an experimental stage and thus, beyond the scope of this paper.

- `first-fit` - allocates the first RDMA memory block that is large enough
- `next-fit` - this is a variant of `first-fit`. It allocates the first RDMA memory block that is large enough, but remembers the last allocation and starts from that
- `best-fit` - Searches the entire list for allocation and selects either the exact or best match
- `worst-fit` - Searches the entire list for allocation and allocates the largest block with respect to the request

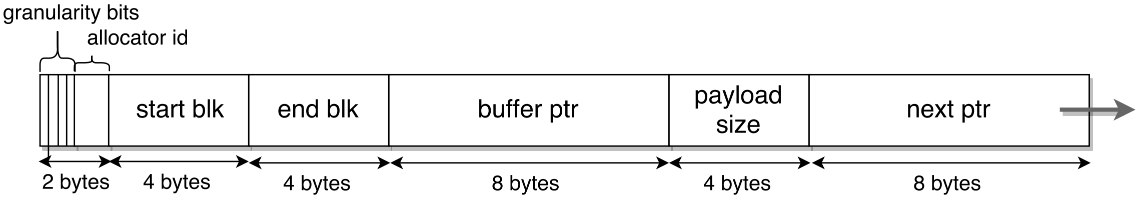


FIGURE 4.2. **A *rmalloc* descriptor (30 bytes)**

4.2. Remote Memory Allocator

The distributed allocator returns special descriptor objects (Figure 4.2) to track all RDMA-enabled virtual memory segments, including ones that are not physically mapped in shadow instances. Each descriptor contains either local or remote boundary tags and other useful information (i.e. actual memory location, payload size, and a granularity mask) for RDMA memory that are *committed* to an RMALLOC network operation. To guarantee contiguous allocation for next-fit like heuristics, RMALLOC handles the overflow state by splitting the allocation request into two – a dummy, followed by an allocation from the start of the segment. Any special alignment requirements for platform specific operations are appropriately handled (i.e. MPI handles this internally, but 4 or 8-byte alignment for special uGNI

FMA_PUT_W_SYNCFLAG, etc). Each RMALLOC dynamic memory instance may contain a list of storage classes (i.e. granularity bits) for fine-grained segregated memory allocation. Currently, an allocation unit allows up to 2G (i.e. $2^{31} \cdot \mathit{unit\_size}$ bytes) allocation units. The role of `rfree()` is to release a *committed* memory block back to an allocator. Thus, *uncommitted* memory (i.e. RDMA memory that is not *actively* used and free) is stored in a sorted doubly-linked list for ease of coalescing. When `rfree()` frees an object, it can immediately coalesce the object with a $O(\log(n))$ look-up and $O(1)$ update time.

RMALLOC implements synchronization protocols for replicating the state of remote and shadow allocator instances. The base algorithm finds a sufficiently large contiguous set of free blocks and then updates the remote state with an RDMA GET operation. To ensure valid allocations are returned at all times, each remote RMALLOC instance carries two different states for local and sync *updates*. Here the local state of remote allocator is modified only when sync updates are transmitted to the peer. Since default synchronization is in Network *deferred* mode, states are synchronized only when allocation limits or a *watermark* are reached. In *immediate* synchronization mode, state exchange occurs for each allocation via a secondary RMALLOC channel. The cost of synchronization for the latter mode is more expensive, but necessary for replicating a fragmented memory state between the remote and shadow instances. We describe the algorithm for synchronization in Algorithm 2. The fragmented state of a RDMA memory allocator instance depends on the allocation heuristic and order of memory release requests completed. A further discussion on this is presented in the next section(s) of this chapter.

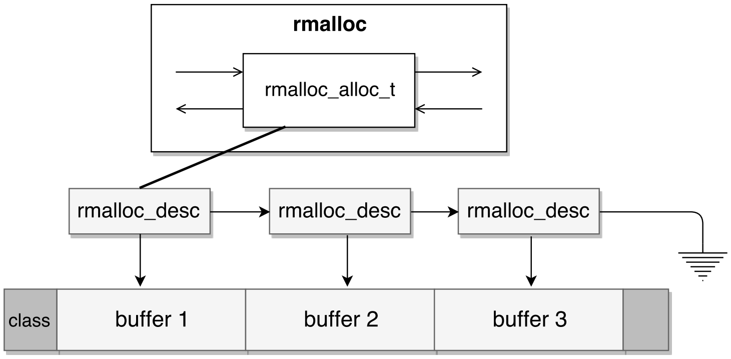
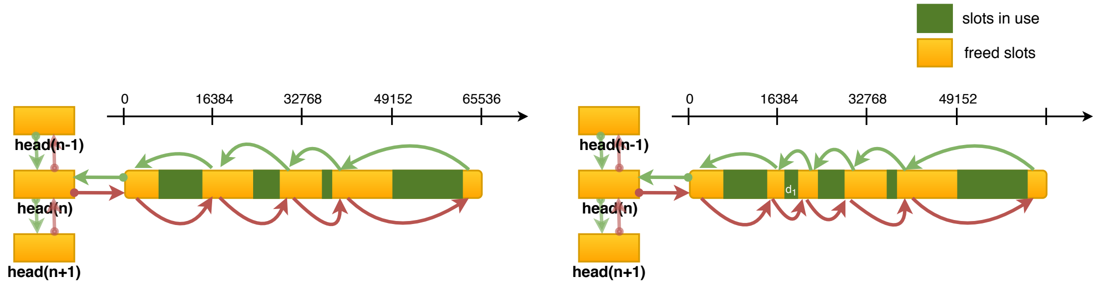


FIGURE 4.3. **Internal Structures for *rmalloc* remote dynamic allocator**

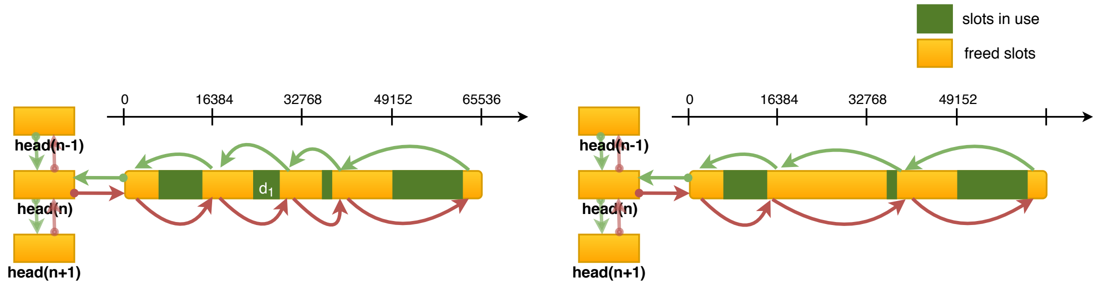
rmalloc is a distributed memory version of `malloc` in Unix based operating systems. However, some of the characteristics of *rmalloc* are more akin to slab allocators [**20, 74**], where segregated allocation allows space and time savings, simpler design (replaces complex trees, bitmaps and sorted coalescing based schemes), and less external fragmentation. Each *rmalloc* dynamic allocator may contain a list of storage classes (i.e. granularity bits) of type `rmalloc_allocation_t`, which are equivalent to slab in a segregated allocator (as shown in Figure 4.3). Every allocation unit (i.e. `rmalloc_allocation_t`) only allows up to a power-of-N allocation with 2G (2^{31}) being the maximum allocation. Apart from the provisioning role, *rmalloc* may also initiate necessary synchronization when the allocation limit is exceeded (which will be described in greater detail in the next section).



a) A fragmented allocator state - before rmalloc(d_1)

b) After rmalloc(d_1)

FIGURE 4.4. **Splitting with explicit free lists in *rmalloc* remote dynamic allocator**



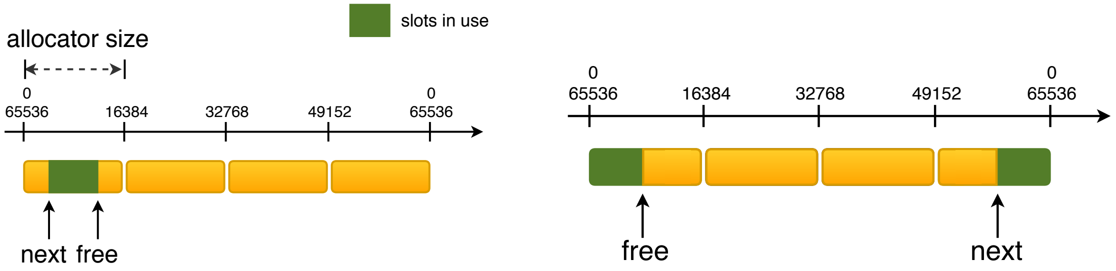
a) A fragmented allocator state - before rfree(d_1)

b) After rfree(d_1)

FIGURE 4.5. **Coalescing with explicit free lists in *rmalloc* remote dynamic allocator**

The "rmalloc" distributed allocator employs special descriptor objects (Figure 4.2 – type `rmalloc_desc_t`), because not all RDMA-enabled virtual memory is physically mapped – this is the case for provisioned `rmalloc()` objects (or memory) for a remote endpoint. Each "rmalloc" descriptor contains local or remote boundary tags and other useful information (i.e. actual memory location, payload size, and granularity masks) for RDMA memory that are *committed* to an "rpipe" network operation. It points to a *start* and *end* block, as well as the actual memory location (8 bytes) and the payload size. Additionally, it contains metadata headers to store allocation granularity as a bit mask and other information, like an allocation identifier. The payload size information is used by an "rpipe" to infer that the polling flag is offset for remote side completion.

For *uncommitted* (freed blocks – RDMA memory that is not used currently), "rmalloc" descriptors are always stored in a sorted, doubly linked list. Each free list also maps to one of the size classes for the segregated lists. For example size classes can be power of two or some arbitrary order. `rfree()` releases an acquired memory block back to an allocator. For every `rfree()` request, an allocator instance inserts the request in the tag order by matching descriptor *start* and *end* tags. Then, the allocator instance will attempt to merge any possible requests in an optimistic manner. If any of the requests are merged, they are removed from the queue and returned back to a corresponding *rpipe* to initiate synchronization. Figure 4.4 and Figure 4.5 show the logical allocator instance state during RDMA memory allocation and de-allocation. For simplicity, Figures only report a single free list (i.e. $head(n)$) for all free or de-allocated blocks. For each allocation request, RMALLOC searches the corresponding size class (i.e. $req <= size\_class(n)$) and finds the possible match from a free region. As shown by Figure 4.4, this will cause a split in the free block and therefore all pointers are updated properly to reflect the split. It is important to note that this action may move the free block to a smaller size class if it does not fit the current. This allows faster searches in segregated allocation. Similarly, `rfree()` may coalesce regions in the same free list (As shown by Figure 4.5) or move them to a larger sized (class) free list after coalescing. These state transitions in free lists also satisfy the property of free blocks are sorted (i.e. by memory region $addr(blk_1) < addr(blk_2) < addr(blk_3)$) and no two are contiguous – if contiguous regions are detected they will always be coalesced. For example RMALLOC handles all cases where either predecessor or successor region of a free block or both are merged accordingly.

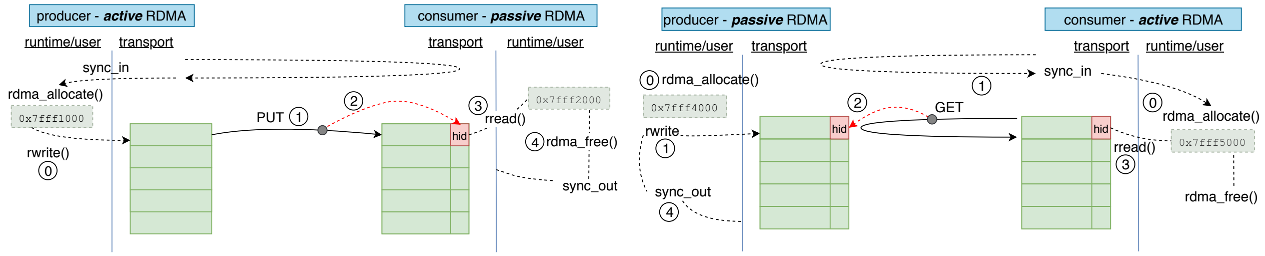


(A) ***no overflow*** (B) **overflow**

FIGURE 4.6. **possible allocator states for `rmalloc()` with `next-fit`**

Figure 4.6 report the special case where the allocation algorithm dictates that memory regions are always allocated from the last known position (i.e. next-fit). While segregated free lists are still maintained for faster searching and coalescing, a circular buffer implementation is utilized for efficiently provisioning of RDMA memory. In this case, RMALLOC is able to implement optimizations to detect full memory capacity and next allocations by direct pointer comparison and arithmetic. However, extra precaution is taken in circular buffer overflow where memory region returned might not be contiguous (Figure 4.6b). For these special cases additional checks are implemented and measures taken to return a valid memory region if possible (i.e. injecting a dummy region and related synchronization actions for remote memory).

The *rmalloc* allocator exports a buffer initialization interface to *rpipe* that allows virtual memory pages to be prepared for RMA transaction. For example, in uGNI back end, interface returns memory that satisfies special alignment requirements specific to Cray Aries/Gemini [**7, 95**] operations (i.e. special 4 byte alignment for RDMA Get and 8-byte alignments for special FMA operations). However, for MPI back end, we directly use the `MPI_Win_allocate` routine, which is MPI specification-friendly and leads to portable integration with MPI-2.2/3.0 APIs.



(A) ***Push* RDMA with completion events**

(B) ***Pull* RDMA with completion events**

FIGURE 4.7. **A Comparison of Event Driven, Low-Level RDMA Control Flow for Symmetric RDMA Modes**

4.3. Symmetric *Push* and *Pull* RDMA

Native RDMA protocols are inherently asymmetric. However, as presented by Figure 4.7, we recognize four distinct stages that are present in both RDMA operation types.

- rdma_allocate – This stage relates to provisioning RDMA enabled memory in remote or shadow instances anticipating a remote RDMA Put, Get, or derivatives operation.
- rdma_free – invoked when data access completion is indicated by either a consumer or producer process that corresponds to `rread` and `rwrite` actions respectively.
- synchronize remote – executed when a shadow allocator state is updated with a remote instance.
- synchronize local – executed when a local instance communicates its state updates with a remote shadow instance.

Within the four common stages that are present, the aforementioned actions will be combined by the transport library to export a symmetric interface for RDMA operations. For example, for active *push* RDMA, allocator state is synchronized on the local site by the consumer process, while for active *pull* RDMA the direction is reversed. Furthermore, remote write for *push* RDMA will execute the RDMA PUT operation, while for *pull* RDMA, a remote write does not initiate any network operation – the action will only prepare the compute buffer for a potential active RDMA GET operation by the peer consumer process. To unify these seemingly different mechanisms in *push* and *pull* RDMA, algorithms separate network actions from memory actions. This separation of concerns allows RMALLOC to build distributed protocols/algorithms that unify both *push* and *pull* RDMA operations as well as, feature efficient synchronization mechanisms.

4.4. Distributed Allocation Algorithms

Algorithm 2 RMALLOC_BASIC(P, V, sz, h)

```
Input: Channel $P$, Allocator instance $V$, allocation size $sz$, heuristic $h$ is the algorithm that generates next allocation,
Output: A descriptor d if successful else ADDR_NULL
Local: Let $(P_b, V_b)$ be secondary channel and $R_{\text{COUNT}}$ num retries.
if ALLOC_ISREMOTE(V) AND SYNC(V) ≡ DEFERRED
  do
    $R_{COUNT} \leftarrow R_{COUNT} - 1$
    d ← FIND_NEXT_ALLOCATION($V, h, sz$)
    if d ≡ ADDR_NULL then
      $s$ ← SYNC_NETWORK($P$)
      UPDATE($V, s$)
      BREAK
  while ($R_{COUNT} > 0$)
if ALLOC_ISREMOTE(V) AND SYNC(V) ≡ IMMEDIATE then
  s ← RREAD($P_b, sizeof(d)$)
  UPDATE($V, s$)
  d ← FIND_NEXT_ALLOCATION($V, h, sz$)
else
  d ← FIND_NEXT_ALLOCATION($V, h, sz$)
  if ALLOC_ISLOCAL(V) AND SYNC(V) ≡ IMMEDIATE then
    $e$ ← RMALLOC($P_b, V_b, sizeof(d)$, nextfit)
    $*e$ ← d
    RWRITE($P_b, e$)
return d
```

Algorithm 2 lists the simplified distributed algorithm for allocating memory between two peers. The RMALLOC distributed allocation works by first finding a relevant allocation via an input heuristic, and then updating the remote state with a zero-copy RDMA operation. Our implementation employs a few well-known heuristics for allocation. The `next-fit` heuristic allocates the first RDMA memory block that is large enough, but saves the state of the last allocation for subsequent requests. The `best-fit, worst-fit` allocation searches the free list for allocation and selects either the exact one or a match that is almost large or small enough for the request. We report two executions for `rwrite()` by Figure 4.8(a) and

Algorithm 3 RMALLOC_SYNC(P, V, sz, h, T)

```
Input: Channel $P$, Allocator instance $V$, allocation size $sz$, heuristic $h$ is the algorithm that generates next allocation,
Output: A descriptor d if successful else ADDR_NULL
Local: Let $(P_b, V_b)$ be secondary channel, $R_{\text{COUNT}}$ num retries, $T \in PUT, GET$ is a RDMA network operation type.
if ALLOC_ISREMOTE(V) AND SYNC(V) ≡ DEFERRED
  do
    $R_{COUNT} \leftarrow R_{COUNT} - 1$
    d ← FIND_NEXT_ALLOCATION($V, h, sz$)
    if d ≡ ADDR_NULL then
      if h ≡ NEXT_FIT AND T ≡ PUT then
        $s \leftarrow$ SYNC_IN_BLOCKS($P$)
      else if T ≡ PUT then
        $s \leftarrow$ SYNC_IN_FREE_TAGS($P$)
      else if h ≡ NEXT_FIT AND T ≡ GET then
        $s \leftarrow$ SYNC_IN_BLOCKS($P$)
      else
        $s \leftarrow$ SYNC_IN_ALLOC_TAGS($P$)
      UPDATE($V, s$)
      BREAK
    else if h ≠ NEXT_FIT AND T ≡ PUT then
      SYNC_OUT_ALLOC_TAGS($P, d$)
    if d ≠ ADDR_NULL then
      BREAK
  while ($R_{COUNT} > 0$)
if ALLOC_ISREMOTE(V) AND SYNC(V) ≡ IMMEDIATE then
  s ← RREAD($P_b, sizeof(d)$)
  UPDATE($V, s$)
  d ← FIND_NEXT_ALLOCATION($V, h, sz$)
else
  d ← LOCAL_ALLOC_SYNC $< T >$ ($V, h, sz$)
  if ALLOC_ISLOCAL(V) AND SYNC(V) ≡ IMMEDIATE then
    $e \leftarrow$ RMALLOC($P_b, V_b, sizeof(d)$, nextfit)
    $*e \leftarrow$ d
    RWRITE($P_b, e$)
return d
```

Figure 4.8(b). We assumed RMALLOC `next-fit` allocator (network) and `best-fit` (immediate) for each case, respectively. As indicated by the figures, both *network* and *bootstrapping* RMALLOC instances may stall until the state synchronization step is completed, however, in the former case, this happens at the last possible moment. Usually, `next-fit` allocators are configured for *network*, while other heuristics may be for either of the modes. Due to the fact that RMALLOC is a distributed allocator between two endpoints, a synchronization

Algorithm 4 LOCAL__ALLOC__SYNC$< PUT >$ (P, V, sz, h)

```
Input: Channel $P$, Allocator instance $V$, allocation size $sz$, heuristic $h$ is the algorithm that generates next allocation,
Output: A descriptor d if successful else ADDR__NULL
Local: Let $(P_b, V_b)$ be secondary channel and $R_{\text{COUNT}}$ num retries.
if ALLOC__ISLOCAL(V) AND SYNC(V) ≡ NONE OR h ≡ NEXT__FIT then
    d ← FIND__NEXT__ALLOCATION($V, h, sz$)
    UPDATE($V, s$)
else
    do
        $s$ ← SYNC__IN__ALLOC__TAGS($P$)
        UPDATE($V, s$)
        d ← FIND__NEXT__ALLOCATION($V, h, sz$)
    while ($d \neq$ ADDR__NULL)
return d
```

Algorithm 5 LOCAL__ALLOC__SYNC$< GET >$ (P, V, sz, h)

```
Input: Channel $P$, Allocator instance $V$, allocation size $sz$, heuristic $h$ is the algorithm that generates next allocation,
Output: A descriptor d if successful else ADDR__NULL
Local: Let $(P_b, V_b)$ be secondary channel and $R_{\text{COUNT}}$ num retries.
if ALLOC__ISLOCAL(V) AND SYNC(V) ≡ DEFERRED then
    do
        d ← FIND__NEXT__ALLOCATION($V, h, sz$)
        if d ≡ ADDR__NULL then
            if h ≠ NEXT__FIT then
                SYNC__IN__FREE__TAGS($P$)
            else
                SYNC__IN__FREE__BLOCKS($P$)
    while ($d \neq$ ADDR__NULL)
else
    d ← FIND__NEXT__ALLOCATION($V, h, sz$)
return d
```

protocol is implemented to replicate the state of remote and shadow instances. A RDMA channel relevant to the active RDMA process always maintains a remote shadow instance, since all allocation decisions on the remote are accomplished by a synchronizing state in these two instances. For example, for specific operations of RDMA PUT and RDMA GET, two allocator instances are maintained on the producer (i.e. PUT) and the consumer (i.e. GET) processes respectively. A RMALLOC allocator will check the state of the local instance consisting of an actual virtual memory region and a remote (shadow) instance, which needs

Algorithm 6 rfree_sync$< PUT >$ (P, V$, d, h$)

```
Input: Channel $P$, Allocator instance $V$, allocation descriptor $d$, heuristic $h$ is the algorithm that generates next allocation,
Output: return 0 if successful else 1
Local: Let $s$ be blocks released by rfree.
if alloc_isLocal(V) AND sync(V) $\equiv$ Deferred then
    if h $\equiv$ NEXT_FIT then
        s $\leftarrow$ free_allocation_blocks$(V, d)$
        sync_out_blocks$(P, s)$
    else
        free_allocation$(V, d)$
        sync_out_free_tags$(P, d)$
else
        free_allocation$(V, d)$
return 0

```

Algorithm 7 rfree_sync$< GET >$ (P, V$, d, h$)

```
Input: Channel $P$, Allocator instance $V$, allocation descriptor $d$, heuristic $h$ is the algorithm that generates next allocation,
Output: return 0 if successful else 1
Local: Let $s$ be blocks released by rfree.
if alloc_isLocal(V) AND sync(V) $\equiv$ Deferred then
    if h $\equiv$ NEXT_FIT then
        s $\leftarrow$ free_allocation_blocks$(V, d)$
        sync_out_blocks$(P, s)$
    else
        sync_out_alloc_tags$(P, d)$
        free_allocation$(V, d)$
else
        if alloc_isRemote(V) AND h $\neq$ NEXT_FIT then
            sync_out_free_tags$(P, d)$ // Shadow allocator (active side) should free up
        free_allocation$(V, d)$
return 0

```

Algorithm 8 Sync_in_blocks(P)

```
Input: Channel $P$,
Output: return number of blocks retrieved if inbound synchronization operation was successful else 0
Local: Let $ac$ be a pointer to network buffer that accumulates blocks, $b_{prev}$, $b_{now}$ indicate number of blocks before and after sync operation
    $ac \leftarrow$ get_network_sync_buf$(P)$
    $b_{prev} \leftarrow$ get_blocks$(P, ac)$
    NETWORK_SYNC_IN$(P)$
    $b_{now} \leftarrow$ get_blocks$(P, ac)$
return $b_{now} - b_{prev}$

```

Algorithm 9 SYNC_OUT_BLOCKS(P, s)

```
Input: Channel $P$, $s$ is number of blocks released.
Output: return 0 if Sync out operation was successful else 1
Local: Let $ac$ be a pointer to network buffer that accumulates blocks
   $ac \leftarrow$ GET_NETWORK_SYNC_BUF($P$)
   $*ac \leftarrow *ac + s$
return NETWORK_SYNC_OUT($P$)
```

Algorithm 10 PRODUCE_TAGS_NB(P, d, ctx)

```
Input: Channel $P$, A rmalloc memory descriptor $d$, sync context $ctx$.
Output: return 0 if tag was appended to queue else 1
Local: Let $Q$ be bounded RDMA queue for tags, $l$ be the size of $Q$ and $t$ be the tag generated from $d$.
$buf \leftarrow$ GET_NETWORK_SYNC_BUF($P, ctx$)
$t \leftarrow (d.start, d.end)$
$Q \leftarrow$ GET_SYNC_QUEUE($buf$)
$l \leftarrow$ GET_QUEUE_LEN($buf$)
if $l \equiv$ MAX_SYNC_QSIZE then
   while GET_SYNC_STATE($ctx$) $\neq$ SYNC
      return 1
   MARK_TAG_STATE($Q$, INV)
   CLEAR_QUEUE($Q$)
   INSERT_QUEUE($Q, 0, t$)
   MARK_SYNC_STATE($Q$, READY)
   MARK_TAG_STATE($Q$, VALID)
else
   MARK_TAG_STATE($Q$, INV)
   ENQUEUE($Q, t$)
   MARK_TAG_STATE($Q$, VALID)
NETWORK_SYNC_OUT($P, ctx$)
return 0
```

to mirror the local state. A synchronization action is hinted at when there is not enough memory reflected by the shadow allocator. Active process will retry this operation a number of times until a valid allocation is returned or the retry count will be expired. In *immediate* synchronization mode, state exchange occurs for each allocation via a secondary RDMA channel. While this mode is more expensive, it may be more relevant for certain types of use cases such as, irregular access patterns. For heuristics such as `best-fit`, provisioning should even be possible in a fragmented memory stage, and therefore, protocol implementation will be more involved. Since remote buffers may be released (i.e. by `rfree()`) in a different order than the shadow, `rwrite()` or `rread()` may not have access to a valid allocation from

Algorithm 11 CONSUME_TAGS_NB(P, ctx)

```
Input: Channel $P$, sync context $ctx$.
Output: return 0 if new allocation(s) are found to queue else 1
Local: Let $Q$ be bounded RDMA queue for inbound tags, $R$ be unbounded queue of allocation descriptors, $s_{state}$ and $t_{state}$ synchronization and tag states, $l$ be the size of $Q$, $T$ be new tag queue and $d$ be the allocation descriptor generated by each tag from $T$, .
$buf \leftarrow$ GET_NETWORK_SYNC_BUF$(P, ctx)$
NETWORK_SYNC_IN$(P, ctx)$
$s_{state} \leftarrow$ GET_SYNC_STATE$(buf)$
$t_{state} \leftarrow$ GET_TAG_STATE$(buf)$
$Q \leftarrow$ GET_SYNC_QUEUE$(buf)$
$l \leftarrow$ GET_QUEUE_LEN$(buf)$
if $t_{state} \equiv$ INV OR $s_{state} \equiv$ SYNC then
    return 1
$T \leftarrow$ CHECK_NEW_TAGS$(Q)$
$R \leftarrow$ GET_TAG_QUEUE$(ctx)$
for each $t \in \mathrm{T}$
    $d \leftarrow$ CREATE_DESC$(P, t)$
    ENQUEUE$(R, d)$
if $l \equiv$ MAX_SYNC_QSIZE then
    MARK_SYNC_STATE$(Q, \mathrm{SYNC})$
    OFF $\leftarrow$ OFFSET_SYNC_STATE$(Q, buf)$
    NETWORK_SYNC_OUT_OFFSET$(P, buf, off)$
return 0
```

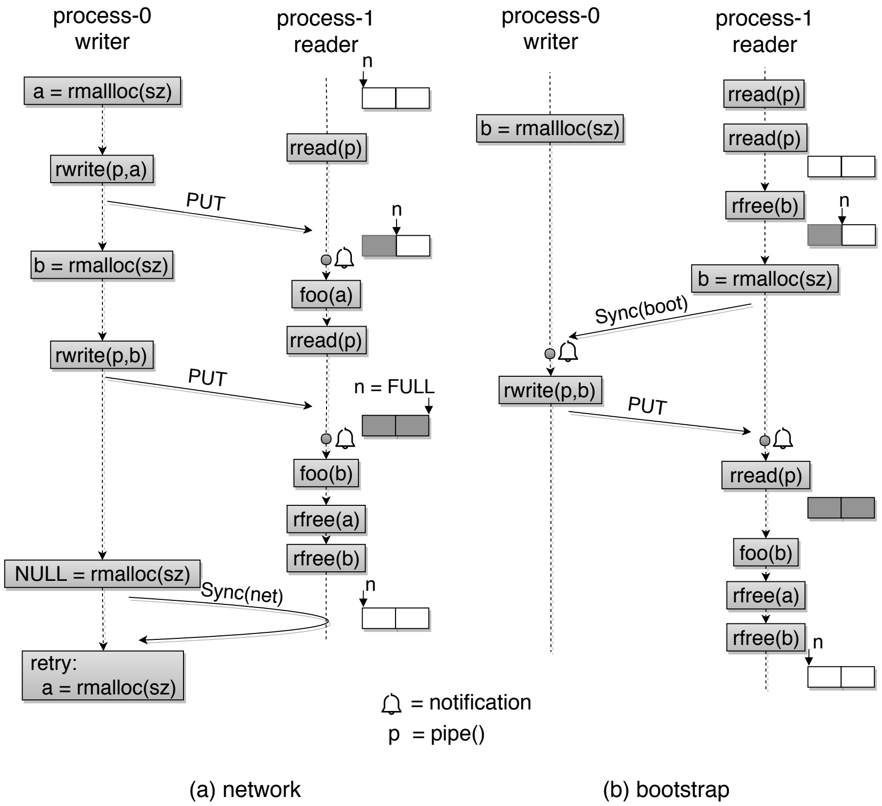
the remote allocator. To ensure proper allocations are accessed at all times, each remote RMALLOC instance carries two different states for local and sync *updates*. Here, the local state of remote allocator is modified only when sync updates are propagated to the peer. The sync operations are implemented using RDMA GET and AMO operations.

In general, RMALLOC attempts to keep synchronization overhead as small as possible. Therefore flow control for allocation instances have been made configurable to fit particular allocation heuristics better. The exact protocols with which these circumstances are handled is presented in Algorithm 3. The default synchronization is still in *deferred* mode, in which states are synchronized only when allocation limits or some *watermark* is reached. For `next-fit`, this means a rendezvous between remote and shadow to exchange number of blocks freed to update `next` and `free` pointers on each side. However, as explained in Chapter 3, state synchronization is handled either by accumulating allocation blocks or by individual tag descriptors. For the case of next-fit, shadow instance is synced by block method for both

RDMA PUT, RDMA Get, and derivatives (i.e. *atomics*). For other non-trivial allocation schemes (i.e. best-fit, worst-fit, etc), this scheme may not work because allocation sizes and ordering may not be known ahead of time for any two communicating processes. Thus, for RDMA PUT operations, the shadow instance heap is expanded by fetching (i.e. `sync_in`) any allocation tag descriptors that were released by the remote instance. Furthermore, once a valid allocation descriptor is returned, it will be pushed (i.e. `sync_out`) to the outbound tag descriptor queue for the consumption of remote instance. For RDMA GET operations, `sync_in` communication is executed on the remotely allocated tag queue. Unlike PUT operations, GET requires fetch from an allocation queue, because RDMA fetch may only be consumed after data is written by the remote process. This synchronization constraint is satisfied by `sync_in_alloc_tags()`.

All types of `sync_in_*` functions are implemented by active messages. This means that blocks or tag descriptors are fetched from a RDMA bounded queue as and when they become available. Thereafter, allocation or release requests for RMALLOC are processed. All instances will also be synchronized with memory acquire or release operations after active messages were retrieved. Shadow allocation routines, as well, as local routines (i.e. Algorithm 4 and Algorithm 5) have enabled this active message synchronization. Algorithm 4 reports the procedure for memory provisioning for local instance(s). For example, in a RDMA PUT, a remote instance first fetches a list of local allocation requests from the active process's outbound tag queue before a message is consumed (i.e. `rread()`). This is because the memory location on which a RDMA PUT operation was executed[2] is only known by the active process. It is important to note that this synchronization constraint will be valid for any heuristic other than next-fit. For RDMA GET, however, the allocator instance is synchronized by tags or blocks that are freed by the passive process as reported by Algorithm 5. The reasoning for this is that the synchronization direction flips for RDMA GET – an active `rread()` operation has to wait until data is written by the producer process.

[2] RDMA PUT, in this scenario, may have already been completed or could still be in flight



(a) network

(b) bootstrap

Figure 4.8. **A possible Execution of `rwrite` with *rmalloc* network or *bootstrapping* channel synchronization**

4.4.1. Releasing Allocated Memory. RMALLOC needs to consider both remote and local instance state whenever releasing (i.e. `rfree()`) RDMA memory back to the allocator. The generic assumption for relinquishing any memory region is that data on the respective memory were successfully consumed before the release operation. However, for RMALLOC , additional constraints also apply depending on which heuristic is employed and whether or not the operation takes place for a RDMA PUT or GET network transfer. As listed by Algorithm 6, for an active process of a RDMA PUT, the library will return the memory immediately back to the allocator instance. However, in the RDMA target process, `rfree()` invokes an additional outbound synchronization procedure. The aforementioned invocation also checks for the type of heuristic and selects whether or not to push either blocks or the allocation descriptor tags back to the queue. Algorithm 7 refers to the `rfree()` operation when the synchronization direction flips in a RDMA GET network transfer. In this scenario, the process that produces the data (i.e. passive `rwrite()`) pushes a number of free blocks or the allocation descriptor tags back to the outbound queue. It is important to note that the `rfree()` action in a RDMA GET transfer is semantically equivalent to the local completion of data (i.e. A write to buffer was complete). Thus, the action enables RDMA active process to `sync_in` any allocation requests that are ready to be consumed. Furthermore, along the same line of reasoning, shadow instance may `sync_out` any consumed allocation requests back to the queue. This enables local instances to synchronize with freed up memory regions (of a shadow), if RDMA memory has been exhausted for an allocation request (cb. Algorithm 5).

4.4.2. Block and Tag Synchronization. Evaluating the efficiency of these protocol implementations warrants a closer look at block-based and tag-based synchronization mechanisms. While both methods serve a similar purpose, they employ distinct algorithms for achieving a synchronized state between shadow and local memory instances as reported by distributed Algorithms 9, 8, 10 and 11, algorithms for block synchronization are straightforward. In `sync_out_blocks`, the producer accumulates a number of blocks released in a network RDMA buffer dedicated for synchronization. While in `sync_in_blocks`, a remote consumer process fetches from the target RDMA *sync* buffer into its local buffer portion. As listed by 8, the consumer tests for new blocks discovered by comparing the previous value in the local buffer. The description of `sync_*_blocks` non-blocking routines also expose the possibility of implementing it, in true one-sided RDMA manner with any capable network fabric. For example, for MPI "back end" `sync_*_blocks` may be implemented with two-sided MPI_ISend/IRecv, while uGNI and/or Infiniband RDMA. However, with one-sided implementation, there exists a greater possibility of communication and computation overlap. Thus, synchronization overhead is generally lower, because of the emergent properties of sparseness and higher throughput provided by these algorithms.

Synchronization algorithms for tags (Algorithms 10 and 11) are designed to operate on multiple contexts. When using non-trivial heuristics (i.e. best/worst-fit), allocated and free memory state of shadow and remote instances could vary significantly, because of

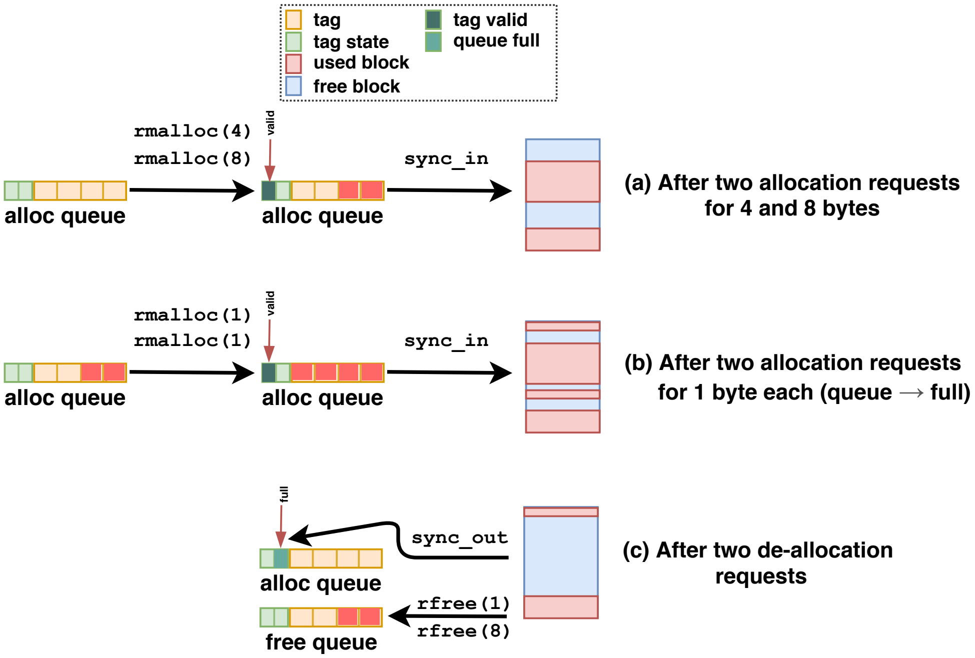


FIGURE 4.9. **A possible execution of *tag* based synchronization for rmalloc**

`rmalloc()/rfree()` ordering. Furthermore, unlike contiguous allocation, shadow instances may not be able to derive which memory regions to acquire or release in a deterministic manner. Hence, multiple contexts allow RDMA channels to track the synchronization state in each direction during `rmalloc()` and `rfree()`. In general, synchronization contexts are supported by creating distinct RDMA memory queues for each context. Each context region may encompass both the tag/sync state, as well as the tag data. Algorithm 10 lists the synchronization protocol for outbound tags. The algorithm first acquires the corresponding queue from the input context and then, creates a tag from the input RMALLOC descriptor. Next, the queue state is made invalid immediately before appending the tag to the queue and made valid immediately after.

Above sequence of actions avoids stale data being consumed at the target process. Since each context queue is bounded, the protocol also checks for a case where the queue is full and thus, waits until synchronization state is restored. As reported by Algorithm 11, the target consumer process checks for any invalid data just after executing a network RDMA `sync_in` operation. If all checks pass, then the discovered tags are appended into a RDMA channel descriptor queue for later processing. Finally, when the inbound queue has consumed up to its total capacity, the consumer process signals the originating process to continue (i.e. that may be waiting as shown by Algorithm 10:*line*:9) by a direct network (RDMA) write to the sync-state (Algorithm 11:*line*:20). Figure 4.9 shows a possible execution of RMALLOC with tag based synchronization. At first two allocation requests are submitted for RMALLOC library. The library consumes these request by appending corresponding allocation tags to RDMA *alloc* tag queue and then marking queue state as valid. The remote allocator instance is then able to execute `sync_in` routine to acquire current allocated tags and segregated memory state is updated accordingly. If the request traffic exceeds the capacity of the queue, RMALLOC then marks the queue as full. It also temporarily postpones acceptance of any further allocation or de-allocation requests until remaining queue items are consumed by the remote instance(i.e. `sync_out`). The remote instance may also free some of the RDMA memory on the remote process. The tag protocol will push such free tags into an outbound RDMA queue (i.e. *free* queue) as shown by Figure 4.9(c) to be consumed later by the shadow memory instance for the purpose of reclaiming memory for future allocations. It is also important to note that, the aforementioned protocol procedures are implemented to be non-blocking and return immediately, if no progress is made. These non-blocking implementations in transport management layer help multiple RDMA channels be processed simultaneously.

4.5. uGNI Implementation

The GNI API by Cray includes two sets of function calls. The user-level, high-performance applications use uGNI functions, while kernel-level drivers may use an API set called kGNI functions. RMALLOC is a user space transport library that primarily relies on uGNI for its “uGNI” network back end. The functionality of the uGNI set of function calls, focuses on their direct interaction with the NIC for high performance RDMA. RMALLOC uses the uGNI API to accomplish the following tasks in order to establish communication among its instance:

- Establish a communication domains and attach it to an NIC device
- Create completion queues (CQs)
- Register RDMA ready memory for use by the NIC
- Create logical endpoints to establish RDMA communication channels
- Use different RDMA modes and synchronized RDMA network events for data transfer and fine-grained completion events.

4.5.1. A uGNI API Overview. The Cray Gemini and Aries interconnect [**7**] are the current state-of-the-art networking hardware designed for the Cray XE, XK, and XC genre of supercomputers. There are two hardware features of Gemini and Aries NIC for initiating network communication: a Fast Memory Access (FMA) unit and the Block Transfer Engine (BTE) unit. To achieve maximum performance, it is important for developers to properly utilize these two distinct components of the Aries and Gemini NIC. GNI implements FMA through a set of memory windows that enable data to be moved by the processor directly from user space through the ASIC to the network. Store instructions into an FMA window are used to generate remote memory reference requests. FMA is primarily used for the efficient transfer of small, possibly non-contiguous blocks of data between local and remote memory. BTE functionality, which is implemented on the ASIC, is intended primarily for large, asynchronous data transfers between nodes. More time is required to set up data for a regular transfer than for an FMA transfer, but once initiated, there is no further involvement by the processor core. BTE transactions can achieve the best computation-communication overlap, because the responsibility of the transaction is completely offloaded to the NIC.

For users to interface with Cray network hardware, two sets of APIs have been developed by Cray: User-level Generic Network Interface [**105**] (uGNI) and Distributed Memory Applications (DMAPP). DMAPP is a communication library which supports a logically shared, distributed memory programming model (DSM) mostly catered towards PGAS run times.

Alternatively, uGNI is designed for applications whose communication patterns are both message passing and remote access in nature. Naturally, we chose to target our network "back end" on uGNI. uGNI provides Completion Queues (CQ) as a lightweight event notification mechanism for middleware to gain both local and remote side completion semantics. While the uGNI API offers rich functionality for supporting communication, it is evident that there are many considerations in terms of performance, overhead, and scalability when selecting the best feature for efficient RMA.

The following APIs are relevant to our uGNI "back end" implementation:

- `GNI_CqCreate()` – Creates a Completion Queue (CQ), which can be linked to an end point or a region of memory for incoming messages. The information of the next event is returned by calling GNI CqGetEvent to poll the CQ.
- `GNI_MemRegister()`, `GNI MemDeregister()` – Memory registration and deregistration. In Gemini/Aries, RMA memory can be used in communication, only after the memory is registered.
- `GNI_PostFma()` – Executes a data transaction (PUT, GET, or AMO) by storing into the directly mapped FMA window to initiate a series of FMA requests. SInce FMA transactions write local buffers into a local window in NIC, the buffer immediately becomes reusable upon return from PostFMA. This is true for Gemini NIC, however, deadlock prevention logic on the Aries NIC requires that the source buffer remain untouched until a corresponding global completion event is received so that the contents of the source buffer may be present (by the NIC), if necessary.
- `GNI_PostRDMA()` – Posts a descriptor to the RDMA queue for BTE in order to execute a data transaction. The descriptor contains all information about the transaction, such as destination process, memory address, handler, etc. The NIC proceeds to execute the transaction in the background (i.e. asynchronously).

An *rpipe* establishes a communication channel for direct RMA operations within two endpoints. A high-level look at the pipe is shown in Figure 3.4. The most basic pipe primitive is a "half-duplex" channel from source to destination process where a RMA transfer only happens in one direction. Alternatively, a “full-duplex” channel may allow RMA transfers in both directions. Other complex pipes such as *“redux”* and *“scatter”* may also be constructed by aggregating primary pipe constructs. Underneath each *rpipe* is one or more *rmalloc* instance and a network “back end” that performs all network specific workhorse RMA network operations on the interconnect (cf. Chapter 3). Each *rpipe* is initialized, lazily, by a synchronized call to `rinit()`. Apart from the common network management of buffer and key exchange routines, this routine may invoke a sequence of actions specific to each network “back end”. For example, uGNI invokes CQ/Endpoint creation and binding routines (i.e. `GNI_CqCreate`, `GNI_EpCreate`, `GNI_EpBind`), as well as buffer registration on the host network adapter with `GNI_MemRegister`.

Algorithm 12 RWRITE$< PUT >$ (P, V, h, d_l)

```
Input: Channel $P$, Allocator instance $V$, local allocation descriptor $d_l$, heuristic $h$ is the algorithm that
    generates next allocation
Output: return 0 if RDMA PUT was successful else 1
Local: Let $d_r$ be a remote allocation descriptor, $ps$ be payload size, $V_l$ and $V_r$ be local and re-
    mote(shadow) allocator instances respectively.
    $V_l \leftarrow$ GET_LOCAL($P$)
    $V_r \leftarrow$ GET_REMOTE($P$)
    $ps \leftarrow$ GET_PAYLOAD($V_l, d_l$)
    $d_r \leftarrow$ RMALLOC_SYNC($P, V_r, ps, h, GET$)
return NETWORK_RDMA_PUT_WSYNC($P, V_l, V_r, d_l, d_r$)
```

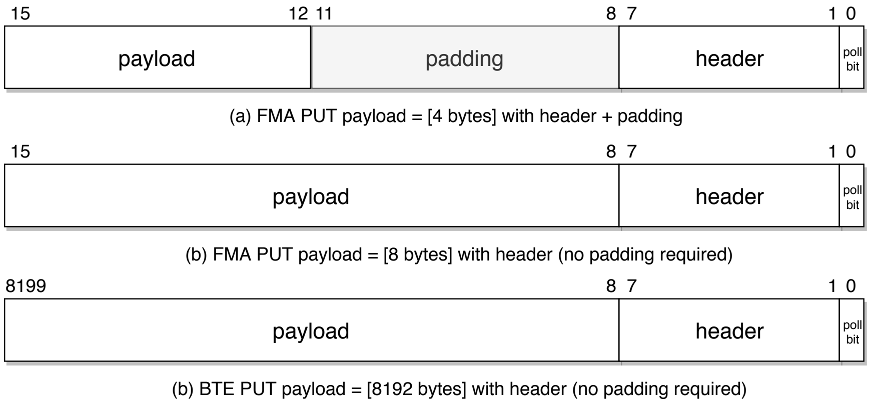
Algorithm 13 RREAD$< GET >$ (P, V, h, d_l)

```
Input: Channel $P$, Allocator instance $V$, local allocation descriptor $d_l$, heuristic $h$ is the algorithm that
    generates next allocation
Output: return 0 if RDMA GET was successful else 1
Local: Let $d_r$ be a remote allocation descriptor, $ps$ be payload size, $V_l$ and $V_r$ be local and re-
    mote(shadow) allocator instances respectively.
    $V_l \leftarrow$ GET_LOCAL($P$)
    $V_r \leftarrow$ GET_REMOTE($P$)
    $ps \leftarrow$ GET_PAYLOAD($V_l, d_l$)
    $d_r \leftarrow$ RMALLOC_SYNC($P, V_r, ps, h, GET$)
return NETWORK_RDMA_GET_WSYNC($P, V_l, V_r, d_l, d_r$)
```

4.5.2. uGNI RDMA Channel Read, Write Protocols. With the preliminaries covered, we will now describe the details of *rpipe* operations which are central to the functionality of our library. We summarize the behavior of `rwrite` and `rread` operations below:

- `rwrite` - for a RDMA Put *rpipe*, an uGNI RDMA PUT operation begins from a source to target buffer. This invocation may also execute a local write to RDMA source memory for an already configured RDMA Get *rpipe*. The semantics of this operation is locally blocking – it waits until PUT or STORE completes.
- `rread` - for a RDMA Get *rpipe*, begins an uGNI RDMA GET operation from a RDMA target buffer to source. Similarly, it may wait for reads from the target of RDMA PUT operation at the target process with an already installed RDMA Put *rpipe*. The semantics of this operation are either blocking or non-blocking and each option is then facilitated by the API – it waits until GET or LOAD completes.

We note that no amount of optimization will help achieve better performance unless the network hardware is utilized to accelerate RDMA to the fullest extent possible. Therefore, `rread` and `rwrite` implementation tightly couples with the network "back ends" to extract maximum performance by thoroughly considering the subtleties of the interconnect.



(a) FMA PUT payload = [4 bytes] with header + padding

(b) FMA PUT payload = [8 bytes] with header (no padding required)

(b) BTE PUT payload = [8192 bytes] with header (no padding required)

FIGURE 4.10. **rmalloc layout for uGNI FMA and BTE transport packets for 4, 8, 8K byte payloads.**

Rpipe uses a specialized hybrid form for notifications called "notified flags with polling" , thus, all local completion events are handled via the events received by a Local Completion Queue (LCQ) and remote completion by polling. We avoid using remote CQs for several reasons. While they provide a popular lightweight notification mechanism [**13, 73**], this may consume significant resources at the network hardware level. CQs are implemented in hardware, thus, it is not just a software abstraction. Moreover, middleware should explicitly manage completion events to avoid overrun events and stale events, thus incurring some software bookkeeping overhead in the process. To enable notified completion semantics efficiently via CQs, transaction events such as completion events with “immediate” data are useful [**13, 50**]. However, such notifications may turn out to be a limited resource for modern network adapters[3]. For use cases like active messages that require much wider metadata matching semantics, messaging layers need to implement extra infrastructure on top of CQ “immediate” data events. We believe polling to be a more generalized alternative. It is important to note that polling can be made highly efficient with the support of synchronized transactions available in Cray Gemini/Aries. Polling also implies that *rmalloc* instances need to provision headers along with the payload that will be written by polled flags during network operations. We keep a (polling) header of 8-bytes and/or 4-bytes depending on the execution mode. Figure 4.10 shows the layout of the RMALLOC transport packet used for uGNI FMA and BTE transactions for Cray Aries (i.e. Cray-XC30 and Cray-XC40) architecture. Each packet contains a section for data payload and header for meta-data and polling bits. The layout of a packet is structured appropriately to maintain byte alignment constraints as required by uGNI [**3**]. For example, RMALLOC transport layer has to handle 8-byte alignment for FMA PUT transactions by inserting additional padding when required (See Figure 4.10). Such constraints differ depending on each RDMA transfer mode or on architecture such as Cray Aries or Gemini interconnect.

For `rwrite`, two types of RDMA put operations are supported: a) FMA PUT with a sync flag; b) Two PUTs - FMA/BTE PUT followed by FMA sync PUT. The motivation behind the former is simply that one PUT is less costly than two PUTs and imposes less contention on the network back end. However, a regular PUT operation (i.e. `GNI_POST_FMA_PUT`) may not be[4] suitable for this task. This is because the receiver is unaware of whether or not the header arrives before the actual payload has. By default, Aries/Gemini instructs the router to adaptively route transactions on a per-packet basis[5]. Moreover, because *rpipe*

[3] For the Gemini adapter, “immediate” data is only a 4-byte integer

[4] However a special case exists for less than 64-bytes of payload, which is the maximum transferable unit (MTU) for Aries

[5] Packet delivery mode can be configured at runtime, but may result in sub-optimal performance

does not implement remote side completion events, there is no direct knowledge of the data having been received. Therefore, *rpipe* uGNI "backend" implements a special *atomic* FMA transfer `GNI_POST_FMA_PUT_W_SYNCFLAG`, which is FMA PUT transaction followed by a special synchronization flag [7]. In addition to sending the regular data, FMA PUT with sync flag sends a user-specified, 8-byte flag delivered to a user-specified offset of the target memory when the operation is complete. For all other cases, we use two PUT operations where the sync flag is sent once and the initiator receives `GNI_CQMODE_GLOBAL_EVENT` at the local CQ. The `rwrite` version for GET operations remote writes poll flags at the source process to indicate that the buffer is in "ready-to-read" state. This action may initiate an actual GET for `rread` on the source process.

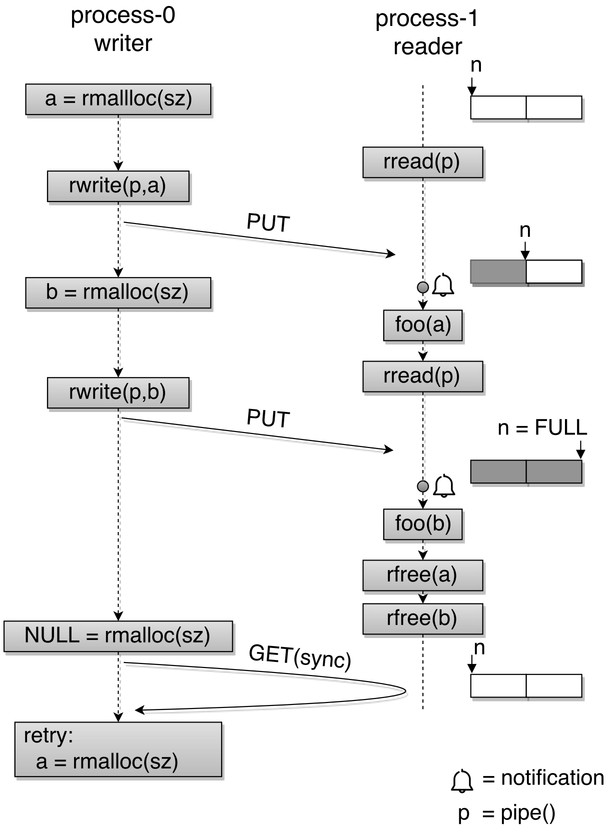


FIGURE 4.11. **A possible execution of `rwrite` on *rpipe* (of size $n = 2$) with *network* deferred synchronization**

A pipe read (i.e. `rread`) may also use two types of RDMA GET operations: a) FMA GET with AMO; b) A FMA/BTE GET followed by a FMA sync PUT. The latter is strikingly similar to the corresponding rwrite, where "notified flags with polling" is accomplished by the completion of GET operation with a sync flag by a GNI_POST_FMA_PUT. Still, the first operation type warrants a closer look. The uGNI library does not have an equivalent single GET operation (as with PUT above), which writes a sync flag immediately after data. Therefore, to accommodate single GETs for small messages (i.e. up to 4KB), rread uses an *atomic* fetch operation (i.e. AMO). As its name suggests, a side effect of FMA AMO is to perform an operation synchronously as soon as the data is delivered. We use a 32-bit single operand fetching XOR AMO operation to notify the target. The consumer process invoking rread fetches data only after the producer process has set the polling flag. Then, immediately after the fetching FMA AMO operation, the target flag is set to zero[6] by the XOR operation. As such, the notification can be detected by testing for zero in poll headers. It should be noted that uGNI "back end" has to attend to adapter specific details when using FMA Sync and AMO transactions. For Gemini Fetching (GET) AMOs requiring the local address, remote address and length need to be 8-byte aligned, whereas Aries does not have such constraints for single operand fetching AMOs[**105**].

Two possible execution scenarios for rwrite and rread are reported by Figure 4.11 and Figure 4.12. The *rmalloc* next_fit allocator is assumed on both instances, which keeps both next (n) and free (f) pointers to track the allocations. Both rwrite and rread are synchronizing operations – depending on the context, they will wait (i.e. blocking) or test (i.e. non-blocking) until sufficient memory is available again or a particular remote operation has been completed. As shown in Figure 4.11, "notified flags with polling" is used to determine if data is available for the consumer upon completion of remote PUT events. Similarly, a fetching consumer (Figure 4.12) of a pipe (i.e. reader or process that issues a remote GET) makes use of "notified flags with polling" to GET data only after it is produced.

4.5.3. CQ Synchronization. uGini FMA/BTE network "back end" requires to implement local CQ handling to properly manage completion events that occur when message

[6] Note that the XOR operator has the following property, $a \oplus a = I = 0$

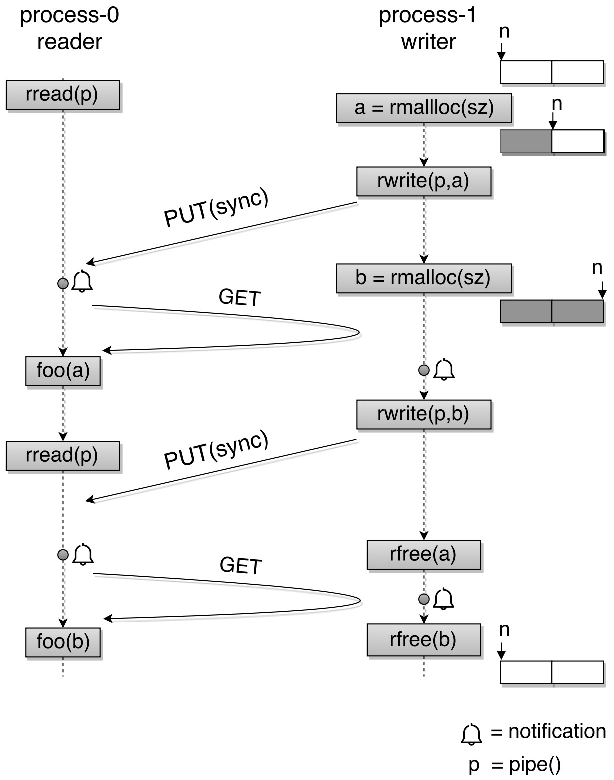


FIGURE 4.12. **A possible execution of a `rread` on a *rpipe* (of size $n = 2$) with *network* deferred synchronization**

is transferred successfully to the destination. If completion events are not managed properly, then it is possible to overrun the network CQs that will lead to an unstable network state. Handling of CQ's may differ significantly, depending on the network type. For example, Cray Aries (i.e. dragonfly) FMA local completion event handlers need to poll for `GNI_CQMODE_GLOBAL_EVENT` to avoid FMA deadlocks. Not doing so may result in GNI invalid parameter errors. However, for Cray Gemini network, this restriction on CQ mode events is not enforced [**3**].

uGNI "back end" configures each endpoint CQ with a fixed number of completion entries. The number of CQ events per channel is tracked by the network handler. A CQ event from free list is acquired whenever a FMA or BTE transaction is executed and then released when a corresponding local completion event is received. A request FMA/BTE network operation is always delayed, if no available completion entries exist in the endpoint CQ.

Each completion event retrieved by GNI_CqGetEvent is matched according to the following criteria: A check is associated with ‘post id‘ for each FMA/BTE request with the portion of the completion event. If this check is successful, then GNI ‘instance id‘ is tested against the endpoint identifier. If both these checks are a success, then both the event and request descriptor are released back to the free list.

4.6. MPI Implementation

A MPI-3.0 RMA compatible window (MPI_Win) is created for each RDMA primary and bootstrap channel in RMALLOC MPI “back end”. This task is accomplished via invocations to MPI_Win_ create_dynamic() and MPI_Win_attach(). Thus, all rread() and rwrite() operations eventually translate to MPI RMA operations on the window. The MPI RMA communication interfaces are comprised of both bulk and fine-grained synchronization primitives for progression and completion (cb. Chapter 2). Currently, RMALLOC MPI “back end” has only selected the fine-grained interfaces as RDMA channels require individual completion events for RDMA operations.

4.6.1. MPI RDMA Channel Read, Write Protocols. We summarize the behavior of rwrite and rread operations below with respect to MPI implementation. The same transport protocols listed in Algorithms 12 and 13 are applied with network operations specific to MPI RMA. However, because MPI RMA exposes a generic higher-level abstraction, some of the control and synchronization operations that were available for uGNI are absent in MPI. Thus, MPI implementation is severely limited in supporting some finer-grained completion events or synchronization support for “notified flags with polling” for improving performance.

- rwrite - for a RDMA Put *rpipe*, begin with a MPI_Win_Put operation on the underlying MPI Window from a source to target. This invocation may also execute a local write to MPI windows source region for an already configured RDMA Get *rpipe*.

- `rread` - for a RDMA Get *rpipe*, begin with a `MPI_Win_Get` operation on the underlying MPI Window. Similarly, it may wait for reads from the target of MPI Win RMA Put operation with an already installed RDMA Put *rpipe*.

MPI RMA does not expose completion events for local or remote CQs. Thus, RMALLOC utilizes combined facilities provided by MPI RMA interfaces to implement required semantics. Furthermore, certain forms of MPI-3.0 RMA synchronization are not generally suitable for RMALLOC communication functions. These include active synchronization functions such as Fence PSCW where a user processes side has to engage in network progression through epochs. A performance efficient implementation for a network "back end" is built around finer grained synchronization events. Such event mechanisms are impossible to implement in MPI RMA active mode as these interfaces are too coarse and two-sided in nature. However, fine-grained synchronization may be supported by RMA passive mode or request based RMA mode [**9, 67**].

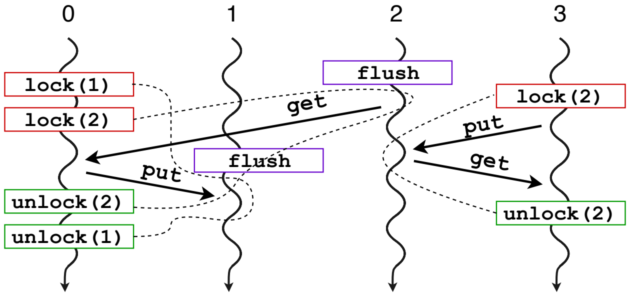


FIGURE 4.13. **Execution of RDMA operations on a MPI "backend" implementation**

When remote completion is required, `MPI_WIN_FLUSH`, `MPI_WIN_UNLOCK`, or `MPI_WIN_UNLOCK_ALL` is called by the MPI network. MPI RMA Locks are used to protect accesses to the locked target window of a RMALLOC RDMA channel, which is affected by RMA calls issued between the lock and unlock calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. Unlike MPI RMA shared locks, accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock-protected.

Thus, exclusive RMA locks maps to a RDMA channels, which is exclusive between two peers. In addition to the aforementioned basic implementation of RMA communication routines, RDMA channels require support for completion flags for `NETWORK_RDMA_GET_WSYNC` and `NETWORK_RDMA_PUT_WSYNC` backing interfaces. Due to the fact that MPI lacks a native mechanism for synchronization flags, `MPI_Win_Put` or `MPI_Win_Get` is followed by synchronization calls in both active and passive processes to guarantee completion. A simplified illustration of these MPI interactions are shown by Figure 4.13, for four "backend" network threads. As shown, a combination of synchronization operations are required along with RDMA operations to guarantee proper execution, which in turn adds up to the overhead as well. On the initiator site (i.e. active), remote MPI Windows offset is calculated first before the follow-up RDMA write takes place. This preparation step is supported by a callback handler to the transport layer, which prepares MPI headers appropriately. MPI_WIN_FLUSH and `MPI_WIN_/LOCK/UNLOCK` actions are sequenced similar to a primary RMA request for the synchronization flags.

4.7. Infiniband Implementation

The Infiniband standard specifies transport services for RDMA based communication. In general, Infiniband advocates for different types of transport services and functions that may be performed on the service. However, not all hardware vendors may support all aspects of the specification, while some others may support only a minimal subset. Thus, capabilities and features may vastly differ depending on the fabric HCA vendor. The following explains the transport services that are defined in the InfiniBand^{TM} specification [**93, 102**]:

- **RC (Reliable Connection)** – Specifies a connection-oriented (i.e. state-full) transport protocol. The connection is also a reliable point-to-point transmission protocol with flow control enabled. Thus, messages are delivered in order, without loss. Due to this fact, reliability guarantees that this mode of transports may have a higher cost associated with it.

- **UC (Unreliable Connection)** – Specifies a connection-oriented transport service without flow control. Packets may be silently dropped during transmission. Thus, this mode of transport may guarantee in-order delivery, but all packets may not be delivered.
- **RD (Reliable Datagram)** – Specifies a connectionless (i.e. state less) transport service without packet loss and flow control. Thus, this mode of transport does not guarantee in-order delivery.
- **UD (Unreliable Datagram)** – Specifies a connectionless transport service that does not guarantee reception or in-order delivery.

All aforementioned Infiniband RC, UC, UD, RD services may be backed up by one-sided RDMA read, write, and atomic operations, as well as loosely two-sided Send/Recv type operations. As noted earlier, the software support for fine-grained operations and QP completion events/notifications may differ by the vendor product, such as Mellanox Connect-X4 or Connect-X5. Variants of Infiniband implementations are also present in Intel and Qlogic fabrics. Similar to Cray and other Infiniband networks, these HCAs may come with different hardware features and the protocol modes available for RDMA. For instance, Intel TrueScale fabric drivers only support send/recv transport functions on connectionless RD and UD services.

4.7.1. Infiniband RDMA Channel Read, Write Protocols. We utilized an Intel native Performance Scaled Messaging(PSM) library for Intel/Qlogic TrueScale (i.e. QLE7XXX Adapters) fabrics to build a reference infiniband “back end” ’implementation for RMALLOC. PSM API [7] exports high-performance, vendor-specific protocols which provide a low-level communication interface for Intel TrueScale and OmniPath architectures. PSM API’s target shared memory and internode modes for communication. Since PSM, in general, carries hardware specific optimizations in its userspace driver library, it exhibits greater performance enhancements for native HCAs than using Infiniband Verbs or a similar library adapted for the Intel driver. PSM2 also follows an endpoint based communication model where one or

[7] PSM comes in two flavours: PSM and PSM2 that was supported on Infini-path and Omni-path family of products respectively. PSM and PSM2 support 64-bit and 96-bit tag formats respectively

more endpoints are initiated per-node before any communication takes place. The initiation phase is loosely coupled into two stages. In the first stage, the user opens an endpoint (psm_ep_open), to obtain an opaque handle for all connect, communication, and progress functions. Just before the second stage of connecting endpoints (psm_ep_connect), the user distributes all relevant endpoint identifiers through an out-of-band bootstrap mechanism. Once the endpoint identifiers are successfully distributed to all processes that wish to communicate, they may be used to connect to the remote endpoint.

Although PSM2 exposes a single endpoint initialization model, it enables various levels of communication functionality and semantics, via its components. Two major components available in PSM and PSM2 are Matched Queues and Active Messages. Matched Queues (PSM MQ) present a queue-based communication model with the distinction that queue consumers may use tags to match incoming messages against a list of posted receive requests (psm_mq_irecv). Active messaging mode (PSM AM) follows the active message semantics where a remote handler is executed from a set of preregistered PSM handler functions. However, AM mode does not expose any direct RDMA buffers or routines. In MQ mode, loose two-sided RDMA semantics are followed for large messages, with the actual transfer being performed after a successful match. Message completion in MQ mode follows local completion and thus, source data is generally safe for reuse. Unlike MPI, PSM requires explicit network operations or implicit conditional operations to ensure network progress. For example , psm_poll, is the most general form of ensuring progress on a given endpoint. Other progress operations such as psm_wait and psm_test may wait or test until progress is made on a single request. Other esoteric forms of progression such as psm_ipeek that functions to query a particular MQ for non-blocking requests that are ready for completion. However, requests ”ready-for-completion” are not generally considered complete by MQ until they are returned to the MQ library through psm_wait or psm_test. Following summarize the core implementation of rwrite and rread for Infiniband:

- rwrite - for a RDMA Put *rpipe*, a PSM MQ mode RDMA Send operation psm_mq_isend, is invoked from a source to target buffer. For remote RDMA writes, depending on the size of the payload, a packet will be sent using either MMIO (*aka* PIO in PSM) or DMA (*aka* SDMA in PSM) methods. This invocation may also execute a local write to RDMA source memory for an already configured RDMA Get *rpipe*.

- `rread` - for a RDMA Get *rpipe*, the operation is executed in a special PSM MQ mode Ready To Send Signal (RTS)/Clear To Send Signal (CTS) protocol. Both RTS/CTS signals and payload are sent on two separate contexts using a 64-bit tag. A source process begins the procedure by sending a RTS control packet once the data is made ready for a RDMA fetch. A RDMA receiver intercepts the RTS signal and send a CTS by posting a `psm_mq_irecv`, on the request context. RDMA GET may eventually start retrieving the data from source once CTS match has occurred – Actual payload transfer may be full filled by a eager send request or a RDMA Get (SDMA) depending on the size of the payload. The semantics of this operation are either blocking or non-blocking and each option is then facilitated by the low level network API.

PSM's high level packet transport layer protocol (IPS-PTL) is responsible for reliably transferring packets over from local endpoint to a remote destination. In receive progress loop, PSM tries to test whether any of the requests are matched with the requests located in the expected queue. If any match was not found then, incoming requests are moved into a special queue called *unexpected* queue. For each receive request, they are again matched against the expected requests. In the case where a natch is not found then incoming request is appended to the *expected* queue. On a successful match or a initiation of a send request, PSM triggers the respective protocol control flow to retrieve or transfer the message.

PSM's PIO and SDMA are distinct low level transport modes for RDMA. For eager messages the payload is generally contained along with the metadata and PIO method is used. In this case sender packs the matching tags and source information and then sends the request using PIO. Sometimes, if the message is too small then a payload is not sent. Instead, data is in-lined with the PIO message header (psm send control block or *scb* header). A PIO request is handled by en-queue‘ing the request descriptor in a PIO send-queue and then triggering an interrupt timer for communicating PIO request to the NIC. Medium sized messages may be handled differently such that the PTL protocol may fragment the message into MTU sized packets for RDMA Send – this guarantees reliable delivery. SDMA method is generally utilized for rendezvous protocol for large message sizes. SDMA too is executed in two stages – a successful match triggers a request to initiate the actual message transfer.

PSM employs two methods for rendezvous RDMA a) Eager Send Back and b) Expected TID (Token ID) transfer. In Eager protocol, the initiator send a RTS control request (with a reference to the initiating request) along with matching metadata. A successful match then triggers a CTS request from the receiver side to initiate sending of the payload. Once CTS acknowledgement is received, the initiator may start transferring payload in MTU packets using PIO. A more efficient method is to switch to an expected TID protocol where initiator sends the token for SDMA buffer along with meta data. In this case the receiver registers an expected TID token get operation for the receiver RDMA buffer and then sends a control packet with requested matching token (sent by the initiator) to start the transfer. However, unlike an RDMA GET, the initiator identifies the location of the data at the target using a "send token" instead of a virtual address. This, of course, assumes that the target has already registered the token and communicated it to the initiator beforehand [8]. Therefore, operation is semantically a two-sided RDMA get.

For SDMA, once the request is received, a DMA transfer is taken place corresponding to the buffer location that will send data from DMA buffer to the NIC interface destined towards the receiver side. The receiving process will execute another DMA transfer from incoming packets to its receive buffers via the DMA engine. However, aforementioned loose two-sided operations are sometimes detrimental to the performance of RMALLOC because "notified flags with polling" relies on direct RDMA with no involvement of RDMA target process. To ameliorate the effects of two sided transfer we pre-post tags as *expected* requests at the earliest possible time such that RDMA transfer may immediately take place. And once a PSM request was progressed successfully (i.e. with `psm_poll/_ipeek` and `psm_test`), we

[8] SDMA actually sends the token as part of the initial MQ message that contains the MQ tag

post these tags back to the *expected* queue. This optimization enables fast RDMA transfers in our PSM network implementation.

CHAPTER 5

Managed RDMA Programming

"It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs."

— Edsger Dijkstra, (1930-2002)

5.1. Semantics

We start by explaining the semantics of RMALLOC in order to better understand the mechanics of RDMA communication via dynamically allocated remote memory. Each distributed RMALLOC instance manages a contiguous segment of virtual memory that belongs to a remote process. Thus, each instance maintains the state of a remote allocator (which we call, "shadow") for the purpose of remote RDMA operations, as well as a local allocator state for initiator side data. We utilize a formal notation to precisely describe the semantic relationships amongst different RMALLOC communication and synchronization operations. We believe this framework is useful to help fully understand the relative cost of synchronization operations among RMALLOC RDMA actions, as well as verify the correctness of RMALLOC programs and usefulness for other tools such as compiler transformations. Importantly, RMALLOC exposes no explicit synchronization operations to the user, except for remote *vmem* instance creation. It assumes a "unified" form for memory consistency where RDMA memory coherency is supported by the underlying hardware[1]. Since RMALLOC allows a fine-grained RDMA completion, our specification follows a relaxed form of consistency (i.e. eventually consistent) for remote memory actions.

[1] Almost all current network interconnect hardware features out-of-the-box hardware coherency for RDMA

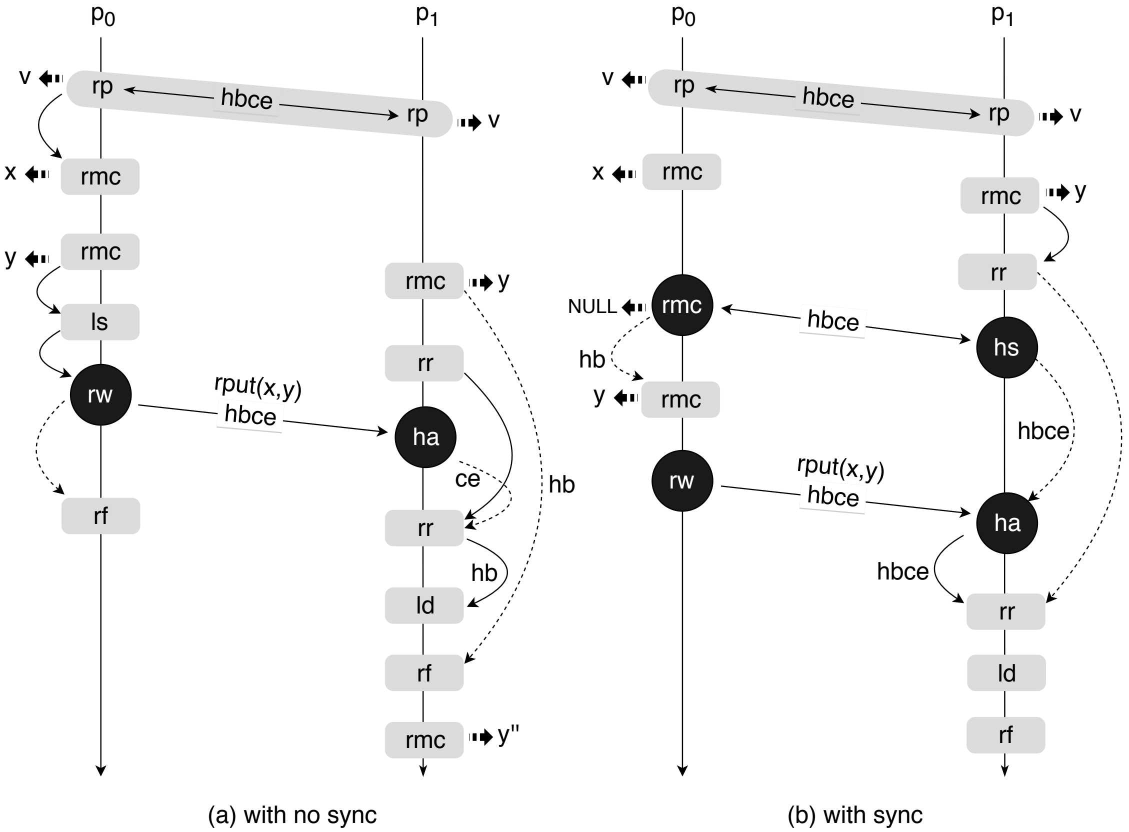


FIGURE 5.1. **A Semantic construction of rmalloc communication and synchronization interactions**

5.1.1. Supported actions. An RMALLOC program consists of two types of operations or actions necessary to define consistency and ordering for one-sided remote events: (a) *memory* actions and (b) *synchronization* actions. The *synchronization* actions may coordinate allocator states between a shadow and local allocator instances to enforce a partial ordering between events. We define the following abbreviation for synchronized RMALLOC instantiation as, `rp`: rpipe. We also define actions on RMALLOC with the following terms: – An action for allocating memory either on the shadow or local *vmem* instance: `rmc`; a *write*-action on the memory returned by a *vmem* instance: `rw`; a *read*-action on the memory returned by a *vmem* instance: `rr`; a local store/load operation on process or instance memory: `ls`, `ll`; and any memory release action: `rf`. All RMALLOC memory actions (`rw|rr`) may translate to a remote RDMA PUT/GET or local memory actions, such as a store or load (`ll|ls`).

Additionally, RMALLOC memory actions (`r*`), as well as allocation/release actions (`rmc|rf`) may generate (See subsubsection 5.1.2.1, subsubsection 5.1.2.2) hidden communication or synchronization actions (`hc|hs`) at the target process.

5.1.1.1. Memory Actions. Following earlier works[**18, 9**], we define memory actions as a 6 attribute tuple.

- **a**: <type> any action of the type (`rmc`), (`r*`) or (`l*`).
- **o**: <origin> originating logical process **id**
- **t**: <target> target logical process **id**
- **wm**: <write memory> the memory address written to by the action (not specified for `ll|rr|rmc|rf`).
- **rm**: <read memory> the memory address read by the action (not specified for `ls|rw|rmc|rf`).
- **am**: <allocated memory> the memory address read by the action by a `rmc` action or `rf` action (not specified for `ls|rw|rw|rr`).

5.1.1.2. Synchronized Actions. All synchronization actions establish a "happens-before" ordering with respect to time. Some of the actions described above generate synchronization effects under certain conditions, so that the intersection of two of these types of actions in a particular execution may not be empty. Each synchronization action in RMALLOC is in fact a 4-tuple defined by $\langle a, o, t, s \rangle$.

- **a**: <type> any action of type (`rp`), (`hc`), (`hs`) or (`rmc`) where *rmc.origin* $\neq$ *rmc.target*.
- **o**: <origin> originating logical process **id**
- **t**: <target> target logical process **id**
- **s**: <sync_mode> contains synchronization mode at the runtime (only specified for `rmc`). Possible values for this field are `immediate`, and `deferred` where a synchronization operation could happen for each request or on-demand.

5.1.2. Ordering, Execution, and Correctness. We assume the existence of two major ordering criteria that guarantee safety and correctness of an RMALLOC program. The first type is the "happens-before" ordering abbreviated as $\xrightarrow{\text{hb}}$. This type of ordering guarantees

P	Program under execution
A	All actions for this execution.
T	Total ordering (transitive closure of $\xrightarrow{\text{ce}} \cup \xrightarrow{\text{hb}}$)
W	Returns the action that produced a particular value in memory for an input action
V	Returns a value corresponding to the memory address entailed by an action

TABLE 5.1. **Description of a `rmalloc` program execution state**

that the relative order regarding time within a single process or two or more processes. The second type of ordering is "consistency-ordering", which we abbreviate as $\xrightarrow{\text{ce}}$. It guarantees relative order for the memory effects on a precedent action, which may take effect eventually for the antecedent action (i.e. eventually consistent). For $x \xrightarrow{\text{ce}} y$, action y may occur in time, well before x, but the memory effects of action x will be guaranteed to be visible to y possibly after an arbitrary amount of time. Understandably, $\xrightarrow{\text{ce}}$ does not provide any timing guarantees (in real time) for $\xrightarrow{\text{hb}}$ relations, however, this type of ordering is essential to describe specific memory actions of an RMALLOC program for polling-based, fine-grained remote completion. We abbreviate certain ordering relations that may guarantee both "happens-before" and "consistency-ordering" as $\xrightarrow{\text{hbce}}$. Furthermore, most actions in RMALLOC may be executed concurrently. Thus, in theory, this kind of time ordering between-actions is specified by $\neg((x \xrightarrow{\text{hb}} y) \cup (y \xrightarrow{\text{hb}} x))$, which we will abbreviate as $x \,\|_{hb}\, y$. An execution is described by a 4-tuple defined as $X = \langle P, A, T, w, v \rangle$, as shown by Table 5.1. Thus, the property: $W(ll|hc) \implies (ll|hc) \cap (V(W(ll|hc)) \equiv V(ll|hc))$ will always be true for a correct data-race free execution of a RMALLOC program.

5.1.2.1. Valid rmalloc Executions. We will now specify valid programs in terms of their executions for RMALLOC . For a program to be safe (race-free, no deadlocks) and correct (consistent behaviour and expected output) at the same time, all possible executions must be valid. An RMALLOC execution is valid under the following conditions:

(1) Each action (excluding hidden communication or sync actions (`h*`)) is created by executing in program order, thus implying $\xrightarrow{\text{hb}}$ and $\xrightarrow{\text{ce}}$ ordering.

(2) A `rp` action ensures synchronization order between two processes. That is, each such action generates $\xrightarrow{\text{hb}}$ and $\xrightarrow{\text{ce}}$ waits-for relationships until `rp` completion. Also, for each $\texttt{rp}_i$ on process i, there must be a corresponding $\texttt{rp}_k$ on another process k, such that,

$$(rp_i \xleftrightarrow{\text{hbce}} rp_k) \cap (rp_i.t \equiv rp_k.o \cap rp_k.t \equiv rp_i.o)$$

(3) All local RMALLOC allocation actions have consistent happens-before relationships with local loads/stores that access the same memory address. That is,

$$\langle (rmc.am \equiv l{*}.{*}m) \cap (rmc.o \equiv rmc.t) \cap (l{*}.t \equiv rmc.t) \rangle \implies rmc \xrightarrow{\text{hbce}} l{*}$$

(4) All remote allocation actions have consistent happens-before relationships with a respective remote put/get. Unlike the MPI-3.0 RMA model, RMALLOC remote put/get actions directly generate hidden communication actions on the remote side, resulting in a $\xrightarrow{\text{hb}}$.

$$\langle (rmc.am \neq r{*}.{*}m) \cap (rmc.o \neq rmc.t) \cap (r{*}.t \equiv rmc.t) \cap (hc.t \equiv rmc.t) \rangle$$
$$\implies rmc \xrightarrow{\text{hbce}} r{*} \xrightarrow{\text{hbce}} hc$$

(5) Each remote write action on process i will be eventually be consistent with the corresponding read action on process k. This enables the user to wait for a message to be completed (via a remote communication action `hc`) without any additional synchronization operations. Thus, an `rr` action that is generated on process k will not have a happens-before relation, however, it will have a `rw` $\xrightarrow{\text{ce}}$ `rr`.[2]

$$\langle (rw_i.wm \equiv rr_k.rm \cap rw_i.t \equiv rr_k.t) \rangle \implies rw_i \xrightarrow{\text{hbce}} hc \xrightarrow{\text{ce}} rr_k$$

(6) RMALLOC remote actions exclusively operate on memory returned by an rmc.[3]. Since `rf` action synchronizes the state between local and shadow allocators, no two `(rr|rc)` actions should operate on the same memory location unless a `rf` has been issued by the user. This means for two $r{*}$ actions,

[2] If process k waits (potentially an unbounded amount of instructions) for the message to arrive by polling on rw.wm, it is guaranteed that the message will eventually arrive.

[3] A provisioning algorithm is implementation-specific, such as first available, `next/best/worst-fit`, etc.

$$\textsc{Let}\, z{\in}(r_0{*}...r_1{*}) \quad s.t. \quad z{:}r_0{*}\xrightarrow{\text{hbce}}z\xrightarrow{\text{hbce}}r_1{*}, \quad \textsc{Then},$$

$$\langle (r_0{*}.a{\equiv}r_1{*}.a) \cap (r_0{*}.{*}m{\equiv}r_1{*}.{*}m) \cap (r_0{*}.t{\equiv}r_1{*}.t)\rangle \implies$$

$$\langle (z.a{\equiv}rf) \cap (z.am{\equiv}r{*}.{*}m) \cap (\forall y{\in}(r_0{*}...r_1{*}){:}(y.a{\equiv}rf) \cap (y.am{\equiv}r{*}.{*}m) \implies y{\equiv}z)\rangle$$

5.1.2.2. Synchronized Executions. RMALLOC shadow allocator synchronizes the state of remote allocator by two modes. In *deferred* mode state, synchronization(by a hidden *sync* action `hs`) is postponed until an instance runs out-of-memory. Alternatively, in *immediate* mode, two states are synchronized at each step.

(1) Following relationships hold for *deferred* and *immediate* modes. We denote shadow and remote allocation actions by rmc_0 and rmc_1, respectively as:

$$\langle (rmc_0.o{\neq}rmc_0.t) \cap (rmc_0.am{\equiv}\Diamond) \cap (rmc_0.s{\equiv}\mathit{deferred})\rangle \implies rmc_0\xleftrightarrow{\text{hbce}}hs$$

$$\langle (rmc_0.o{\neq}rmc_0.t) \cap (rmc_1.o{\equiv}rmc_1.t{\equiv}rmc_0.t) \cap (rmc_0.s{\equiv}\mathit{immediate})\rangle$$

$$\implies rmc_0\xleftrightarrow{\text{hbce}}rmc_1$$

5.2. Programming Model

The basic programming interfaces for RDMA *Managed Buffers* are categorized into five forms, which are listed below. The subsections that follow, also present their common usage patterns. Further details on the APIs and usage information are presented in Appendix B.

- **Setup and Teardown** – `rinit()` and `rexit()` – Library setup routines include both initializing *bootstrap* and *transport* networks.
- **Channel Creation** – `rpipe()` – Creates an RMALLOC remote memory instance between two peers by exchanging RDMA buffers via a *bootstrap* network.
- **Memory Operations** – `rmalloc()`, `rfree()`, `rmalloc_mem()`, and `rmem()` – Acquires peer memory on remote process and release when buffers are no longer needed. The `*_mem()` APIs return actual pointers, instead of virtual descriptors.
- **Network Operations** – `rread()`, `rread_mem()`. `rwrite()` – Performs network operations such as RDMA PUT, GET, and Atomic on top of RMALLOC memory instance

and synchronization. `rread_nb()` and `rwrite_nb()` interfaces allow checking for asynchronous completion of RDMA messages.

- **Process Operations** – `get_rank()`, `get_ranks()`, `get_name()`, `get_nic_addresses()`, `etc` – Reports useful information about the process locality, network and bootstrap information. Additionally, APIs such as `send()`, `recieve()`, `network_barrier()` and `network_allgather()` provide cluster wide synchronization support using bootstrap network.

5.2.1. Zero-Copy Message Passing. Each RDMA *Managed Buffers* channel can be configured by `rpipe()` with a set of local and global parameters that define the nature of RDMA operations allowed (See Listing 5.1).

```
rmalloc_config_t cfg; rch_t ch;
cfg.allocator.peer = peer;
cfg.allocator.rdma_active = is_active_process;
cfg.allocator.unit_size = size;
cfg.allocator.alloc_size = ch_size;
cfg.allocator.op = RDMA_PUT;//_GET OR _ATOMIC
rpipe(ch, cfg);
```

Listing 5.1. **Initializing a RDMA *Managed Buffers* Channel**

The local parameters include the peers, allocation algorithm, total virtual memory required, allocation granularity, and whether or not the current process is associated with active zero-copy operations. Additionally, each channel is configured for a particular RDMA operation type which is either RDMA_PUT, RDMA_GET, or an RDMA Atomic operation type. Global parameters are specific to the underlying network fabric and configure settings such as: the maximum number of completion events allowed, message transfer modes, and message switching points (i.e. from MMIO mode to DMA mode for large messages). It is important to note that each memory channel gives access to direct peer-to-peer RDMA memory locations (via `rmalloc()` and `rmem()`).

Table 5.2. Zero-copy massage passing between processes $i \to j$

addr=rmalloc(rch, 4);
ptr=rmem(rch, addr);
*a = SOME_VALUE;
//RDMA remote write
rwrite(rch, addr);
	rread(rch, addr, 4);
	ptr=rmem(p0, addr);
	comp_val=(*p0)*100 + C
	//release
	rfree(addr);

As such, applications do not have to manage memory by themselves, which is illustrated by Table 5.2. However, once a zero-copy operation is completed, `rfree()` is invoked to return the memory to allow proper synchronization between shadow and remote allocator states.

5.2.2. Halo Exchange. Domain decomposition is common among many parallel applications where a data domain is split over many logical units. A common means of addressing domain-decomposed communication is to rely on halo-cells (or ghost-cells), whose role is to locally mirror a domain residing in a remote process. Halo exchange is at the heart of many scientifically structured grid applications, including PDE/ODE solvers.

```
for (it = 1; it <= STEPS; it++) {
 if (left != NONE)
  rwrite(left_ch[1], rm_lrows_out);
 if (right != NONE)
  rwrite(right_ch[1], rm_rrows_out);

 if (left != NONE) {
  rm_lrows_in_mem =
    rread_mem(left_ch[0], bytes, &rm_lrows_in);
  rm_lrows_out_mem =
    rmalloc_mem(left_ch[1], bytes, &rm_lrows_out);
 }
 if (right != NONE) {
  rm_rrows_in_mem =
    rread_mem(right_ch[0], bytes, &rm_rrows_in);
  rm_rrows_out_mem =
    rmalloc_mem(right_ch[1], bytes, &rm_rrows_out);
 }
 // remote memory update with new data
 remote_update(start, end, &u[iz][0][0],
  &u[1 − iz][0][0], rm_lrows_in_mem, rm_rrows_in_mem,
        rm_lrows_out_mem, rm_rrows_out_mem);
 if (left != NONE)
  rfree(left_ch[0], rm_lrows_in);
 if (right != NONE)
  rfree(right_ch[0], rm_rrows_in);
 iz = 1 − iz;
}
```

Listing 5.2. **2D stencil halo exchange by RDMA *Managed Buffers* transport**

A RDMA *Managed Buffers* transport maps to this communication pattern particularly well, because mirrored domains can be represented by a remote RDMA channel. It facilitates remote memory allocation for multiple halo exchanges all at once for high throughput halo exchanges, which reduces communication overhead between neighbour processes. Listing 5.2 presents a RDMA *Managed Buffers* reference implementation of a halo-exchange. As illustrated, memory channels are created to support exchanges of row data for each left and right direction of the domain. The respective `rwrite()` invocations update halos on the remote memory, while `rread()` will consume the zero-copy values for stencil calculation of the next iteration. Here the stencil application only has to deal with zero-copy operations and does not implement any synchronization, which is handled transparently by the transport library.

```
void handle_hashreq_thread(t_hash* hash){
 int i, flag = 0;
 int64_t pos, offset;
 for (i = 0; i < hash−>procs; ++i) {
  if(i!=hash−>r){// assume remote −−>local read
  t_elem * nreq = NULL; // hash element
  if(lreqs[i] == ADDR_NULL){
    lreqs[i] = rmalloc(rch[i], sizeof(t_elem));
  }
  rread_nb(rch[i], lreqs[i], &flag);
  if(flag){
    nreq = rmem(rch[i], lreqs[i]);
    pos = hashfunc(nreq->value);
    offset = (pos - hash->r*hash->tsize);
    update_element(hash, nreq, offset);
    rfree(lreqs[i]);
    lreqs[i] = ADDR_NULL;
}}}}
```

Listing 5.3. **Active Message Read Routine of a DHT by a Thread on RDMA *Managed Buffers* API**

5.2.3. Active Messages. Active messages [**112**] are extremely useful in asynchronous messaging when receivers are not aware of how many messages to expect or even from which receivers to expect them. Of particular consideration are the data-driven applications and algorithms with irregular communication patterns, such as classes of graph navigation, detection [**81, 39**], graph mining [**40**] or distributed machine learning algorithms. RDMA *Managed Buffers* is well equipped to support this active messages, because of the native event-based nature of the transport. In Listing 5.3, we show an example of where active messages for a distributed hash table read were implemented by RDMA *Managed Buffers* . The request processing thread checks all incoming RDMA channels periodically for an active message; if a message is found, then the request is decoded to update a specific hash bucket in the local portion of the DHT. Any arbitrary number of threads and information may be encoded by the request, including a memory-mapped handler function for asynchronous processing. Due to the fact that intercepting active messages for a channel has to be created between each communicating peer, a quadratic number of channels may be required at start-up (i.e. $O(n^2)$ for n processes) in the worst case. This may potentially lead to a scalability bottleneck with the increasing channel usage for active messages. A possible solution [**72**] to this problem would be to create a few bytes-wide monitoring channel (i.e. 1 to 4-bytes that contain meta-data about channel setup and tear-down) between all endpoints. With this approach, a parallel program may monitor channel events and, based on this information, initializes and retires them on demand.

Algorithm 14 ALGORITHM_EM3D(G, V, E)

```
Input: Graph $G$, Vertices for local domain $V$, Edges for neighbourhood domain $E$
Local: Let $N_e$ and $N_h$ be the neighbourhood vertex set of E and H cells respectively. Let $STEPS$ be the total number of time steps
INIT_EM3D_SYSTEM(G, V, E)
for $i \leftarrow 0$ to $STEPS$
    GATHER_FROM_H_NEIGHBOURS(G, V, E, $N_h$)
    ALL_COMPUTE_E(G, V)
    GATHER_FROM_E_NEIGHBOURS(G, V, E, $N_e$)
    ALL_COMPUTE_G(G, V)
EMIT_E_H(G)
```

5.2.4. Irregular Communication. The exchange of vertex and edge data of a graph will lead to highly irregular communication, if the partitioning domain is asymmetric. In such circumstances, sparsely distributed many-to-many data dependencies are the norm. Synchronization overheads are generally high because each graph communication domain needs to guarantee the safety of successive network RDMA operations between updates to the same vertex or edge set. Additionally, traditional message passing overheads in MPI and MPI-3.0 RMA also degrade performance (See Chapter 1). Consider a scenario where communication converges to a bipartite graph. EM3D is a statically partitioned, irregular bipartite graph problem that models the propagation of electromagnetic waves through objects in three dimensions. It uses the Discrete Surface Integral method [**32**] on an unstructured 3D mesh. In EM3D, an object or 3D mesh is divided into cells that contain a mix of adjacent nodes representing either an E field or H field. The computation consists of a series of integration steps that alternate between half-time steps (i.e. N time steps contains $2 * N$ half-steps). Specifically, the value of each E node is updated by a weighted sum of neighboring H nodes, and then, H nodes are similarly updated using the E nodes.

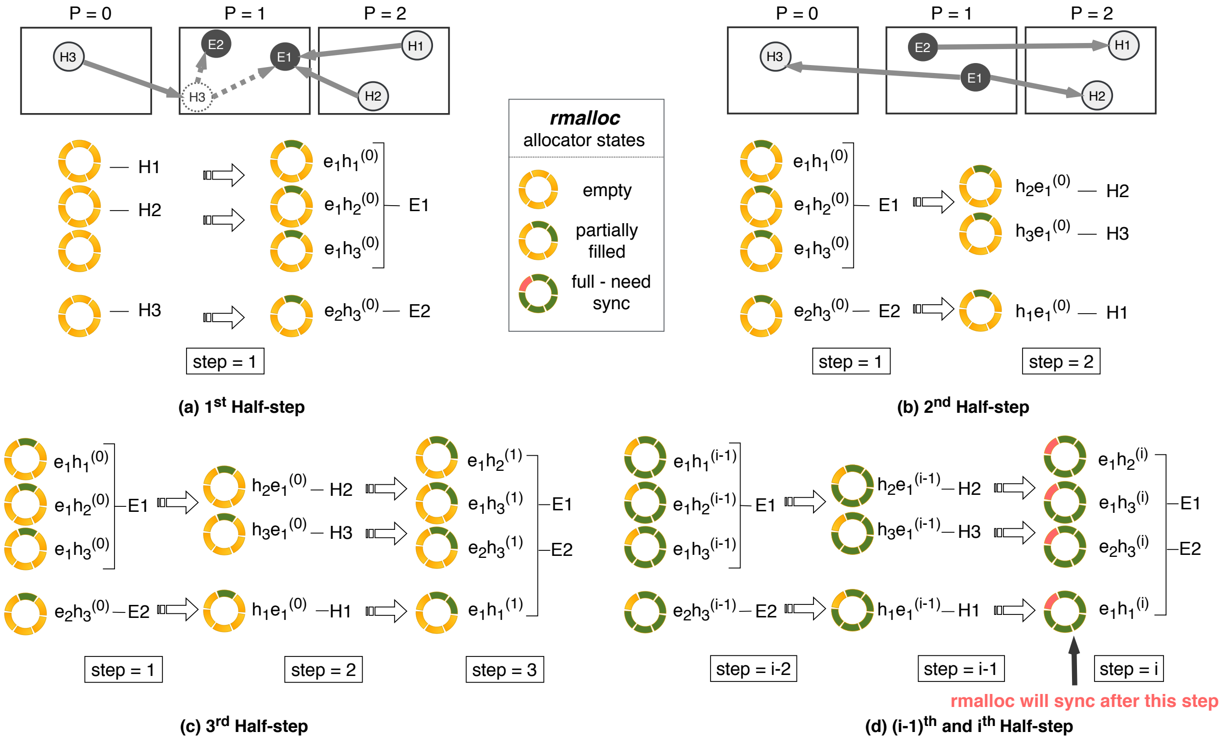


FIGURE 5.2. **RDMA *Managed Buffers* for EM3D – each instance $e_ih_j^{(m)}$ or $h_ie_j^{(m)}$ for ith origin, jth target and mth remote memory address.**

We report the basic parallel algorithm for EM3D (Algorithm 14) and also list respective basic program code for EM3D written in MPI message passing, MPI-3.0 RMA and RDMA *Managed Buffers* for `Gather_from_H_neighbours()` and `All_compute_E()` in Listings 5.4, 5.7 and 5.8. An MPI message passing implementation requires `MPI_ISend()` and `MPI_IRecv` for H to E data transfers in order to compute new E values. MPI guarantees message synchronization when there is both matching and network progression. In contrast, MPI RMA Get and PUT implementations require explicit synchronization – EM3D RMA GET version issues a sync operation to complete all RDMA GETs and a barrier at the end to ensure that all successive RMA GET operations will be made visible in the most recent updates[4]. The barrier synchronization guarantees that no unintended RDMA updates take place before

[4] The EM3D RMA PUT version has a similar global barrier that stalls until all RDMA PUT operations are complete

cell data are computed for each epoch. An important distinction for the RDMA *Managed Buffers* version is that no explicit synchronization operations are implemented in the application code. The operations `rwrite` and `rread`[5] facilitate direct event-driven RDMA operations with either RDMA PUT or RDMA GET on MPI or low-level fabric, such as GNI. Importantly, regardless of the mode of operation, synchronization will be handled by the RDMA *Managed Buffers* transport layer on behalf of the application. Figure 5.2 shows the communication and sparse synchronization steps when EM3D problem is mapped to an RDMA *Managed Buffers* program.

```
for(i=0; i<H_SZ; i++)\\send H cells
 MPI_Isend(H[i].data, .., H[i].r);
\\recv from H neighbours
for(i=0; i<E_SZ; i++)
 for(j=0; j<E[i].e; j++)
  MPI_Irecv(E[i].data[j],&Ereq[i][j]);
MPI_Waitall(Ereq);
for(i=0; i<E_SZ; i++)
 for(j=0; j<e_nodes[i].e; j++)
  update_e(E[i].data[j], E[i].w);
```

Listing 5.4. **Gather_H Compute_E in MPI**

```
for(i=0; i<E_SZ; i++)
\\ RDMA GET from H neighbours
 for(j=0; j<E[i].e; j++)
  MPI_Get(E[i].data[j], ...,
              E[i].rem[j]);
MPI_Win_sync();
for(i=0; i<E_SZ; i++)
 for(j=0; j<E[i].e; j++)
  update_e(E[i].data[j], E[i].w);
MPI_Win_fence();
```

[5] For brevity, we have used a synchronous mode of `rread` in code listings, but asynchronous functions are also supported

Listing 5.5. **MPI3.0 GET**

```
for(i=0; i<H_SZ; i++){ \\ RDMA PUT E cells
 MPI_PUT(h_nodes[i].data, h_nodes[i].rem);
}
MPI_Win_fence();
for(i=0; i<H_SZ; i++){
 for(j=0; j<e_nodes[i].e; j++)
  update_e(e_nodes[i].data[j], e_nodes[i].w);
}
```

Listing 5.6. **compute_E in RDMA PUT**

```
for(i=0; i<E_SZ; i++){ \\ RDMA GET from H neighbours
 for(j=0; j<e_nodes[i].e; j++)
  MPI_Get(e_nodes[i].data[j], e_nodes[i].rem[j]);
}
MPI_Win_sync();
for(i=0; i<E_SZ; i++){
 for(j=0; j<e_nodes[i].e; j++)
  update_e(e_nodes[i].data[j], e_nodes[i].w);
}
MPI_Win_fence();
```

Listing 5.7. **compute_E in RDMA GET**

```
for(i=0; i<H_SZ; i++)
 rwrite(rch[i], H[i].data_out);
\\ recv from H neighbours
for(i=0; i<E_SZ; i++)
 for(j=0; j<E[i].e; j++)
  rread(rch[i][j], E[i].data_in[j]);
for(i=0; i<E_SZ; i++)
 for(j=0; j<E[i].e; j++)
```

```
update_e(E[i].data_in[j], E[i].w);
rfree(E[i].data_in[j]);
```

Listing 5.8. **RDMA *Managed Buffers***

5.3. Collective Programming

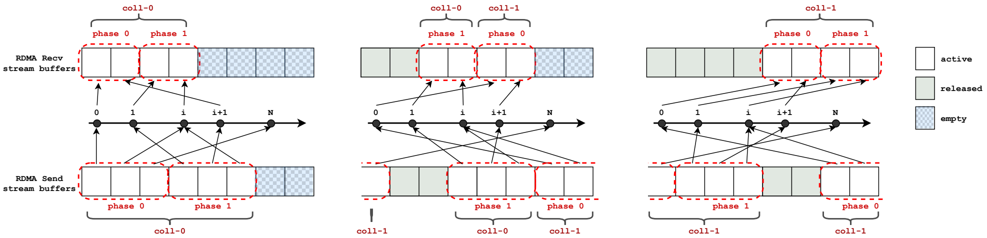


Figure 5.3. **A collective schedule designed with rmalloc RDMA programming**

Collective frameworks, such as LibNBC [**63**] enables an asynchronous progression of group communication primitives to better utilize communication and computation overlap for parallel applications. LibNBC exports a flexible network layer to plug in new byte transport implementations, as well as an internal layer which enables for seamless integration of new collective patterns and operations. The internal layer presents itself in the form of collective schedules – a dependency graph of send/receive operations. RMALLOC programs leverage this capability to encode schedules using RMALLOC objects. There exist few advantages to utilizing RMALLOC RDMA managed programming model in collective schedules. First, it eliminates the need for network initialization, bootstrapping, and RDMA synchronization (i.e. flow control, request tracking, completion, and progression of RDMA transfers) in the framework library – instead, the transport library manages such tasks via RDMA channels created between logical nodes of a schedule. Furthermore, as shown by Figure 5.3, utilizing RMALLOC remote memory allocator instances, each phase iteration of a particular schedule will be mapped to a stream of direct RDMA buffers. A subset of these RDMA buffers will be actively sending or receiving messages (i.e. *active* or *release* states) at any given point in time while others may not have been utilized yet (i.e. *empty*). The streaming state of RDMA channels allows for more communication operations to be overlapped within multiple phases of a single schedule or between multiple iterations of schedules. Therefore, RMALLOC RDMA channels may allow collective schedules to be optimized for high throughput. However, the space required for RDMA channels and the maximum allowed independent overlapping operations within each scheduling algorithm are limiting factors to consider here.

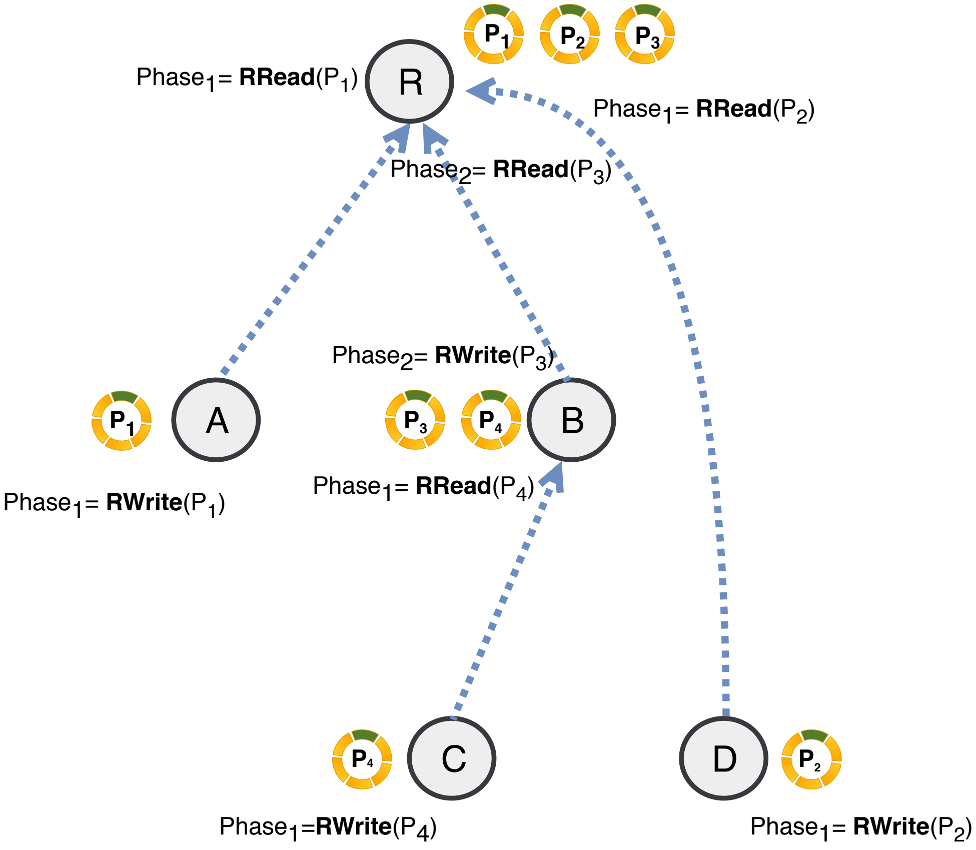


FIGURE 5.4. **A Collective Schedule for REDUCE with a Binomial Tree Topology Created by rmalloc RDMA Channel Instances**

For example, Figure 5.4 represents a collective *reduce* operation running on five logical nodes implemented by four "rpipe" RDMA channels. In this scenario, the communication operations of the reduction are parallelized in two phases using a binomial tree topology. Each phase may be separated by a phase barrier to satisfy data dependencies of the algorithm. A scheduled collective algorithm with multiple phases is transformed into `rread()` and `rwrite()` actions during the initialization time. A Phase barrier is executed by waiting for all RDMA read and write requests to be completed. Importantly, RMALLOC channels will be bootstrapped before any communication takes place. This step is executed during `rcoll_init()`, which parses a given schedule in order to build a Directed Acyclic Graph (DAG) of RMALLOC instances.

```
static int rcoll_NBC_Start_round(rcoll_NBC_Handle *handle) {
  int *numptr; /* number of operations */
  int i, res, ret=NBC_OK;
  NBC_Fn_type *typeptr;
  NBC_Args *sendargs, * recvargs, ;
  NBC_Args_op *opargs;
  NBC_Args_copy *copyargs;
  NBC_Args_unpack *unpackargs;
  NBC_Schedule myschedule;

  myschedule = (NBC_Schedule*)((char*)*handle−>schedule + handle−>row_offset);
  numptr = (int*)myschedule;

  /* typeptr is increased by the size of(int) bytes to point to type */
  typeptr = (NBC_Fn_type*)(numptr+1);
  for (i=0; i<*numptr; i++) {
    switch(*typeptr) {
      case SEND:
        sendargs = (NBC_Args*)(typeptr+1);
        typeptr = (NBC_Fn_type*)(((NBC_Args*)typeptr)+1);
        bytes = sendargs−>bytes*sendargs−>count;
       // allocate remote RDMA memory from RDMA channel
           rch_t rm = rcoll_get_channel(handle, sendargs−>peer, true);
           raddr_t addr = rmalloc(rm, bytes);
           void * m = rmem(rm, addr);
           rcoll_input_cb in_cb;
           if (sendargs−>use_rdma_buffer) {
            // in_cb is collective input handler
            in_cb = handle−>coll_einf−>coll_input.input_cb;
            in_cb(m, bytes, sendargs->tmp_offset, handle->coll_einf->coll_input.args);
            } else {
            in_cb = rcoll_copy_from_tmp;
            in_cb(m, bytes, sendargs->tmp_offset, handle->tmpbuf);
           }
           res = rwrite(rm, addr);
           sendargs=NULL;
        break;
      case RECV:
            recvargs = (NBC_Args*)(typeptr+1);
            typeptr = (NBC_Fn_type*)(((NBC_Args*)typeptr)+1);
                bytes = recvargs->bytes*recvargs->count;
            handle->req_count++;
            handle->req_array = (rm_req_t*)realloc((void*)handle->req_array,
                                           (handle->req_count)*sizeof(rm_req_t));
            NBC_CHECK_NULL(handle->req_array);
            // test for incoming requests
            rm_req_t *curr_req = handle->req_array+handle->req_count-1;
            curr_req->args = *recvargs;
            curr_req->state=REQ_PENDING;
            curr_req->recv_ch = rcoll_get_channel(handle, recvargs->peer, false);
            curr_req->recv_addr = rmalloc(curr_req->recv_ch, bytes);
        if(curr_req->recv_addr == ADDR_NULL){
             ret = -1;
             goto error;
            }
            recvargs=NULL;
        break;
      case OP:
        opargs = (NBC_Args_op*)(typeptr+1);
        typeptr = (NBC_Fn_type*)((NBC_Args_op*)typeptr+1);
        NBC_Operation(handle, opargs->op, opargs->datatype, opargs->count);
        break;
```

```
      case COPY:
        copyargs = (NBC_Args_copy*)(typeptr+1);
        typeptr = (NBC_Fn_type*)((NBC_Args_copy*)typeptr+1);
        res = NBC_Copy(handle, copyargs->srccount, copyargs->srctype, buf2,
                                    copyargs->tgtcount, copyargs->tgttype, handle->mycomm);
      default:
        printf(”rcoll_NBC_Start_round:␣bad␣type␣%li␣at␣offset␣%li\n”, (long)*typeptr,
                                    (long)typeptr−(long)myschedule);
        ret=NBC_BAD_SCHED;
        goto error;
    }
    /* increase ptr by size of fn_type enum */
    typeptr = (NBC_Fn_type*)((NBC_Fn_type*)typeptr+1);
  }
  if(handle->row_offset != sizeof(int)) {
    res = rcoll_NBC_Progress(handle);
  }


error:
  return ret;
}
```

Listing 5.9. **A NBC Scheduler Implementation for rmalloc Collectives**

The two key functions that perform network progression for a collective operation includes scheduling (i.e. `rcoll_NBC_Start_round`) and driving the collective communication progress (i.e. `rcoll_NBC_Progress`) accordingly. Listing 5.9 shows the run time scheduler function `rcoll_NBC_Start_round` for the rmalloc collective implementation. It iterates through each phase of a collective algorithm and performs the appropriate action, as declared by the schedule. For RDMA actions (i.e. Send/Receive), it first acquires the appropriate RDMA channel from the collective handle. If the type of scheduling stage is a RDMA `SEND`, then memory is administered on the peer node via a respective remote allocator instance. A callback function will then be invoked to prepare the input data. One input processing case is then handled depending on whether they originate from a user or through an intermediate RDMA buffer address (i.e. generated by the many phases of a collective algorithm). Once the input processing step is completed, the collective run time performs a necessary RDMA Put/Get network operation. Similarly, if the type of action is a `RECV`, then returned memory from the remote RMALLOC instance is tested for completion.

The network completion and progression phases are executed in `rcoll_NBC_Progress`. Completion routines primarily use RMALLOC non blocking APIs, such as `rread_nb`, because many requests for collective schedules must be progressed at the same time to avoid deadlocks. Once a request is successfully tested for completion, any remaining scheduling stages (i.e. `OP`, `COPY`, `PACK`, etc) are executed. Finally, memory is released back to the remote instance via `rfree`. Listing 5.10 reports a simple parallel reduction application written in RDMA managed programming. A reduction operation is first initialized by a RMALLOC collective configuration. The basic parameters for a configuration include: type of collective operation, data type and count for the group communication primitive, RDMA operation type, and a processing function for reduction type collectives. Additionally, each collective may be configured for the size of RDMA channel allocators supported (i.e. *steps*) and other collective operation specific parameters, such as the root node. The collective implementation also supports non blocking execution for enhanced communication and computation overlap by executing compute functions, while also testing for collective completion.

```
int reduce_input(void *input, size_t sz, int offset, void *args) {
  int *in = input;
  *in       = SOME_VAL;
  return 0;
}

rcoll_params_t p = {
            .type      = REDUCE,
            .count     = 1,
            .datatype = MPI_INT,
            .op       = MPI_SUM,
            .root     = 0,
            .rdma_op  = RDMA_PUT,
            .steps    = STEPS
                };
int main(){
  rcoll_handle_t coll_handle;
  rinit(&rank, &nprocs);
  int ret = rcoll_init(&coll_handle, p, reduce_input, NULL);
  network_barrier();
  for (i = 0; i < ITERS; ++i) {
    while (!flag) {
      rcoll_nb(coll_handle, &flag);
    }
    rcoll_reset(coll_handle);
  }
  rcoll_free(coll_handle);
  return ret;
}
```

Listing 5.10. **An Example Reduction operation executed with RDMA managed buffers interfaces**

CHAPTER 6

Evaluation

"A scientist naturally and inevitably ... mulls over the data and guesses at a solution. He proceeds to testing of the guess by new data-predicting the consequences of the guess and then dispassionately inquiring whether or not the predictions are verified."

— Edwin Hubble, (1889-1953)

We evaluated the reference implementation of RMALLOC with the help of standard benchmarks suites and a selected set of HPC applications. However, the selected parallel applications reflected dissimilar communication patterns. Thus, we believe this selection was ideal for evaluating effectiveness of RMALLOC for different conditions. The reference implementation was comprised of two bootstrap implementations (i.e. PMI and MPI) and three "back end" implementations for MPI, Cray, and Infiniband. For all experimentation, we used either 4 or 8-byte flags for "notified flags with polling" synchronization and a fixed completion event size. RMALLOC was compiled with native optimizations turned on. The gcc 5.0 compiler was used with optimization enabled for all static and dynamically linked libraries and binaries. Most of our standard benchmark suites are executed on two nodes to reflect on our primary objective of evaluating inter-node communication performance. Each measurement was executed in fixed (i.e. 1000 - 10000 depending on the benchmark) iterations in order to accurately represent the cost of a single operation.

To realize our goal of measuring effectiveness of RDMA *Managed Buffers* programming, each application was subjected to the same parallel algorithm and the communication/compute stages for all systems and libraries. Since our focus is on loosely coupled RDMA communication with completion events, comparison was against MPI-3.0 RMA fine-grained completion mode with `MPI_PUT/GET` with *locks* (i.e. `MPI_Win_lock/unlock` and `MPI_Win_flush`) [1] and GASNet active messages. Furthermore, for a baseline comparison, we included regular MPI message passing with RDMA. The micro-benchmarks are based on the Ohio MicroBenchmark Suite (OSU) for MPI two-sided and one-sided operations and the GASNet test suite for direct RDMA events. We used production grade implementations of MPI (Cray-MPICH-2.1.4, Open-MPI-4.0) and GASNet-Exascale(GASNetEx) for all relevant results. We used a reference implementation backed by Cray-uGNI for RMALLOC transport configured with 8-byte synchronization flags. Each benchmark measurement was executed in 100,000 iterations in order to accurately represent the cost of a single operation. For all analysis, we calculate *response time improvement* by $\frac{T_1 - T_b}{T_b}$ and *speedup* by $\frac{T_b}{T_1}$ where T_b is the baseline measurement. All experiments were run on the Dagger, Bigred II, Big Red II+, Big Red III clusters, and supercomputers at Indiana University. Dagger is a small Infiniband cluster connected by Intel/Qlogic QLE7340 TrueScale Infiniband [**35**] single port network running on 16 Intel Xeon E5 cpu nodes. Bigred II consists of a 3D Torus topology with Cray XE6 Gemini network [**95**] running on 344 cpu nodes [2]. Each XE6 node has two AMD Opteron 16-core Abu Dhabi x86_64 CPUs and 64GB of RAM. Bigred II+ consists of a Cray Dragonfly topology on Cray XC30 - Aries network [**7, 3**] with 550 compute nodes. Each has two Intel Xeon E5 12-Core x86_64 CPUs and 64GB of DDR3 RAM. Bigred III has the same topology as II+, but features an upgraded XC40 network with 930 dual-socket compute nodes equipped with Intel Haswell Xeon processors.

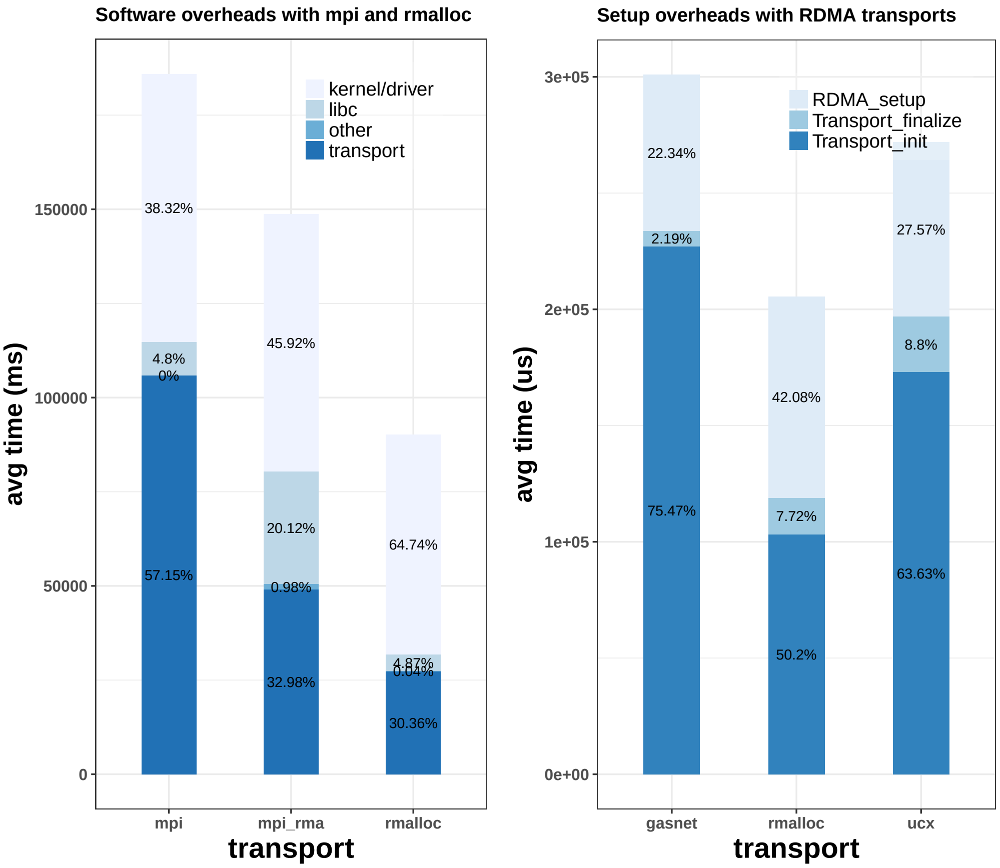
6.1. Performance Benchmarks

Micro-benchmark results [**118, 117**] provide an overview of RMALLOC's capabilities in terms of aspects such as more standard metrics latency, bandwidth, or indirect metrics, such as cost of synchronization. For standard benchmark measurements, we used the benchmark suite already available at Ohio State University, which contained suites of libraries for point-to-point performance (i.e. p2p latency, bandwidth with ISend/Irecv or MPI3.0 RMA), as

[1] We performed our analysis against both *unified* and *separate* modes of MPI-3.0 RMA

[2] Bigred II GPU nodes were not used for any of these experiments

well as collectives. We also implemented new benchmarks with RDMA *Managed Buffers* that closely resembled existing standard benchmark for MPI or GASNet. Other synthetic benchmarks emulated a particular scenario for profiling and was implemented by respective run time (MPI-3.0 RMA) and RMALLOC transport. The latest release of HPCToolkit [**4**] library was employed in some other benchmarks for sampling measurements and profiling overhead in the software run time.



(A) **Breakdown in run time layer for MPI, MPI-3.0 and Rmalloc**

(B) **Breakdown for Channel Setup and Tear-down in RDMA Allocators**

FIGURE 6.1. **A Stacked profile view of average overheads in RDMA software**

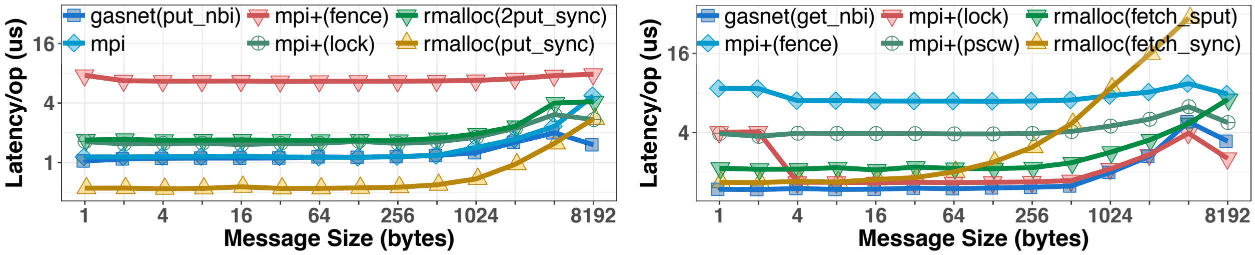
6.1.1. Evaluating Setup and Runtime Costs. The cost associated with run time execution environment may become fixed or dynamic, depending on the RDMA usage. An example of a fixed cost could be setup and tear-down time for RDMA bootstrap. However, the problems in run time execution could be more subtle. This is because control flow for a network operation may take many paths during execution time(cf. Chapter 1). Furthermore, analysing individual costs makes it difficult to discern the origin of bottlenecks. Based on these assumptions, we identified four component layers for further analysis of run time execution. As reported by Figure 6.1a, descriptions for aforementioned categorization are listed below:

- **kernel/driver** – Time elapsed in kernel or user space drivers to initialize and execute RDMA transactions.
- **transport** – Time elapsed for transport management routines and flow control within the library.
- **libc** – Usage of standard C library functions, such as string manipulations, copying, allocation, and locks.
- **other** – Additional software execution costs that may be attributed to dynamic linking, etc

Figure 6.1a shows analyzed run time execution times in the above component layers for a random RDMA PUT operation (medium-sized message 4,000 bytes) executed under a fixed number of iterations. We profiled MPI, MPI-3.0 RMA, and RMALLOC for time spent under each significant subroutine call and aggregated the measurements that fell under each category. All of the executions exercised similar times in network byte transfer layer functions. However, software overhead incurred in the transport components were significantly larger in MPI and MPI RMA (4.8% and 20.12%), respectively. This is justified by the various library functions executed inside MPI message passing protocols, in addition to RDMA (i.e. functions in openMPI [**48, 52**] such as *ob1* in PML MCA lib`mca_pml_ob1`). MPI also spends considerable time in a message passing engine in order to ensure that correct ordering and matching semantics are guaranteed. While the MPI-3.0 RMA instance spends less time than MPI in protocol management, total time was increased by costs associated with MPI Window based RMA synchronization functions (i.e. MPI_Win_lock, MPI_Win_flush). In terms of analyzing static overheads in RDMA, we evaluated three RDMA transports RMALLOC , GASNet, and UCX. We identified three major components that contribute to RDMA static setup time. As reported by Figure 6.1b, a detailed description of the aforementioned categorizations are listed below:

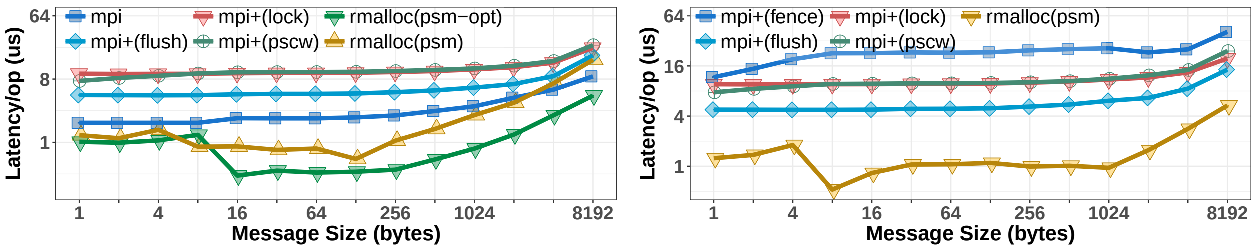
- **Transport_init** – specifies the time elapsed in transport initialization routines. This may be attributed to the software cost of setting up out-of-band networks, event queues, RDMA registration for control regions, connection domains, endpoints, and CQ. It may also contain the overhead involved in initializing structures related to node topology, locality, and utility services.
- **Transport_finalize** – specifies the time elapsed in cleanup routines that gracefully deregisters attached memory regions, registered endpoints and queue, destroys connection domains, CQs, and clears control and auxiliary structures that were created during startup.
- **RDMA_setup** – specifies average time elapsed for creating a communication channel. Main costs involved are attributed to RDMA bootstrap exchange (cf. Chapter 3).

Figure 6.1b reports a breakdown of average setup and tear-down costs associated with RMALLOC , UCX, and GASNet RDMA libraries when a RDMA channel was created between two nodes. We observed that RMALLOC features a relatively lightweight initialization process (i.e. about 1.64X to 2X faster than UCX and GASNet). This is because RMALLOC has a minimalist startup procedure and only initializes global data structures and partially bootstrap which will be completed lazily (i.e. only when creating "rpipe"s). GASNet and UCX perform bootstrap, as well as a more involved system specific setup during startup. Their initialization may execute additional tasks, such as register additional signal handlers to support interrupt-based implementations or aggressive segment registration policies [**19**]. Due to the additional out-of-band setup time, channel creation overhead is about 15-20% more for RMALLOC than the rest. While not a significant factor for usability, we observed that GASNet showcased the lowest tear-down time compared with the rest.



(A) **RDMA PUT** (B) **RDMA GET**

FIGURE 6.2. **RDMA Put and Fetch Latency Comparison for Small Messages in Cray-XC**



(A) **RDMA PUT** (B) **RDMA GET**

FIGURE 6.3. **RDMA Put and Fetch Latency Comparison for Small Messages in Infiniband™**

6.1.2. Measuring Standard Network Characteristics. As reported by Chapter 4, for `rwrite`, two types of RDMA PUT operations were supported for uGNI "back end". First, network hardware assisted FMA PUT with sync flag operation. The second version used is an additional PUT request i.e. FMA/BTE PUT followed by FMA sync PUT. Extra PUT operations (i.e. `GNI_POST_FMA_PUT`) are not generally preferable because tags may arrive out of order invalidating completion semantics[3]. More importantly, direct knowledge of data being received is not possible without some form of notification event or mechanism. In consideration of this, the uGNI "back end" implements a special FMA transfer

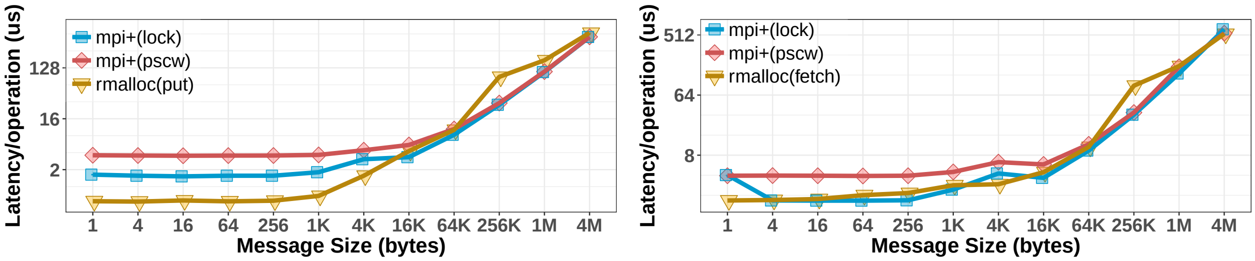
[3] Cray NIC allows packet delivery mode (i.e. in-order delivery) to be configured at run time, but may result in sub-optimal performance

GNI_POST_FMA_PUT_W_SYNCFLAG [**7**]. In addition to sending the regular data, FMA PUT with sync flag sends a user-specified 8byte flag delivered to a user-specified offset of the target memory when the operation is complete. For all other cases we use two PUT operations where the sync flag is sent once and the initiator receives GNI_CQMODE_GLOBAL_EVENT at the local CQ. Figure 6.2a, reports RDMA performance comparison for PUT – rwrite() with synchronization flags having much lower latency numbers. For small- to medium-sized messages, "rpipe" RDMA write that sync PUT is on average 2.7X[4] faster than MPI-3.0 passive RMA and 3X to 6X faster than MPI-2.2/3.0 active pairwise RMA mode and "rpipe" two-PUT rwrite mode, respectively. The optimized rwrite() overall showed 8X and 2X-3X speedup against MPI-2.2/3.0 barrier RMA mode and two-PUT rwrite() (2put_sync) and GASNetEx put_nbi (non-blocking mode), respectively. Figure 6.3a reports latency measurement comparison for RDMA PUT in Infiniband^{TM} . The optimized PSM "backend" implementation showed 1.5X to 6X speedups in latency for rwrite() w.r.t. MPI on small to medium messages. Optimized version for PSM was in average 1.5X faster in rwrite()s compared to the first implementation. The spikes in measurement points in RMALLOC are due to switching of different IPS-PTL (See Chapter 4) transport modes for eager messages – we did not tune RMALLOC for optimal transfer point for this set of benchmarks.

A rread() for uGNI was evaluated with two types of RDMA fetch operations in its transport management implementation. First, FMA GET with Atomic operation called AMO. Second, a FMA/BTE GET followed by an extra FMA synchronized PUT request will be issued (cf. Chapter 4). In Figure 6.2b, we show the results of RMALLOC RDMA read for small/medium messages. FMA GET with AMO provides extremely fast *notified* data access (about 2.5X with respect to MPI-3.0 RMA) for small messages up to 16-bytes. However, because executing a remote XOR operation for each completed network packet is expensive, it does not scale well with message size. Therefore, the transport switches to GET followed by a synchronized PUT (i.e. fetch_sput – "FMA GET followed by FMA PUT"(medium-sized

[4] We calculate speedup by $\frac{T_b}{T_1}$, where T_b is the baseline measurement.

messages-1,000) and to “BTE GET followed by FMA PUT” for large messages(1,000+)) featuring comparable performance [**117, 118**] with `rread` for messages up to 1024-bytes (only 1.25X slowdown, with respect to MPI-3.0 passive RMA, but 1.85X improvement over MPI-3.0 active RMA). GASNetEx uGNI conduit `get_nbi` was the winner by 8-13% speedup for RDMA GET with respect to MPI RMA. In Figure 6.3b, we report the latency measurements for RDMA GET for our reference Infiniband^{TM} cluster. The PSM “backend” implementation showed 2.5X to 6X speedups in latency for `rread()` w.r.t. MPI for small to medium messages. Optimized version for PSM was in average 1.5X faster in `rwrite()`s compared to the first implementation. All observations suggested that RMALLOC PSM implementation surpassed all MPI implementations in terms of latency for RDMA GET. Again, the spikes in measurement were seen in RMALLOC , but were due to switching in different eager IPS-PTL transport modes which was not tuned.

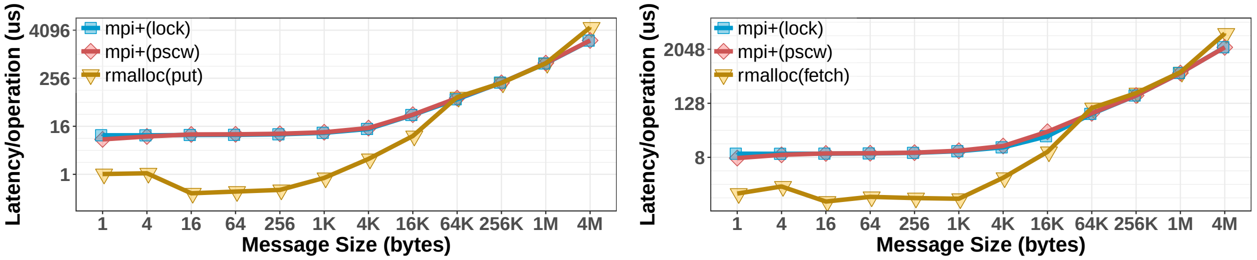


(A) **RDMA PUT**

(B) **RDMA GET**

FIGURE 6.4. **RDMA Put and Fetch Latency Comparison for All Message Sizes in Cray-XC**

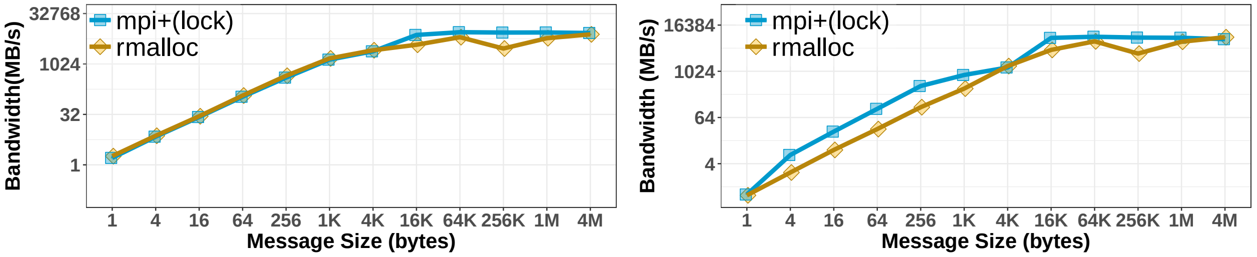
In Figure 6.4a and Figure 6.4b we report latency benchmark results of *rpipe* RDMA write and read respectively for small to large messages in a Cray-XC cluster. The inflection points at 4K messages for pipe `rwrite` benchmark are due to the protocol switch between FMA and BTE. Also for rpipe `rread` data transfer protocol switches from “FMA GET with AMO” (small messages) to “FMA GET followed by FMA PUT”(medium-sized messages) and finally to “BTE GET followed by FMA PUT”(large messages). The switches happen exactly at message sizes 64 bytes and 1K for `rread`. In both cases *rpipe* reads and writes perform better for small and medium-sized messages (up to 4K) and for larger messages



(A) **RDMA PUT** (B) **RDMA GET**

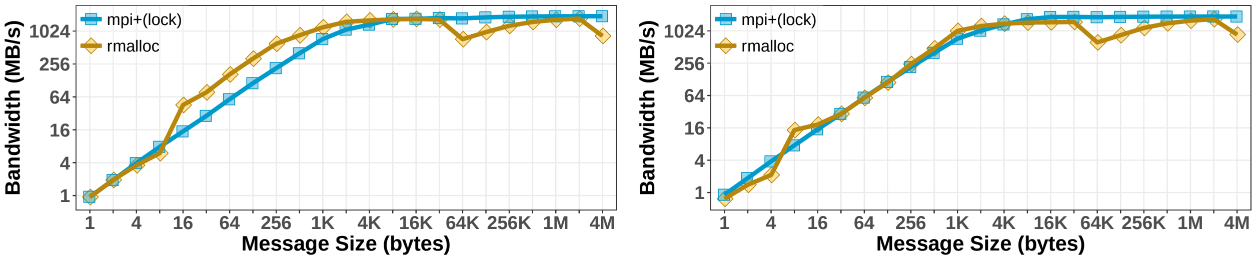
Figure 6.5. **RDMA Put and Fetch Latency Comparison for All Message Sizes in Infiniband^{TM}**

latency measurements per operation were comparable to that of MPI-3.0, with passive mode (i.e. PSCW or fence) being the slowest among the group. The observations after executing same set of experiments in Infiniband^{TM} are reported by Figure 6.5. For messages upto 64K both PSM `rwrite` and `rread` outperformed MPI latency by a range of 3.5X-14X (`rwrite`) and 3X-10X (`rread`) respectively. For larger messages, latency was comparable with MPI implementations. The difference between these observations are primarily attributed to the IPL-PTS switch which made larger messages ($> 64K$) transferred by SDMA (cf. Chapter 4). Another inflection point at 16bytes for RMALLOC PSM was seen as a consequence of two eager mode switches (i.e. *immediate* to *fragment* transfer). PSM optimizations and extra synchronization requests in MPI was reflected in higher sensitivity for small message latency.



(A) **RDMA PUT** (B) **RDMA GET**

Figure 6.6. **RDMA Bulk Put and Fetch Bandwidth Comparison for MPI and rmalloc in Cray-XC**



(a) **RDMA PUT**

(b) **RDMA GET**

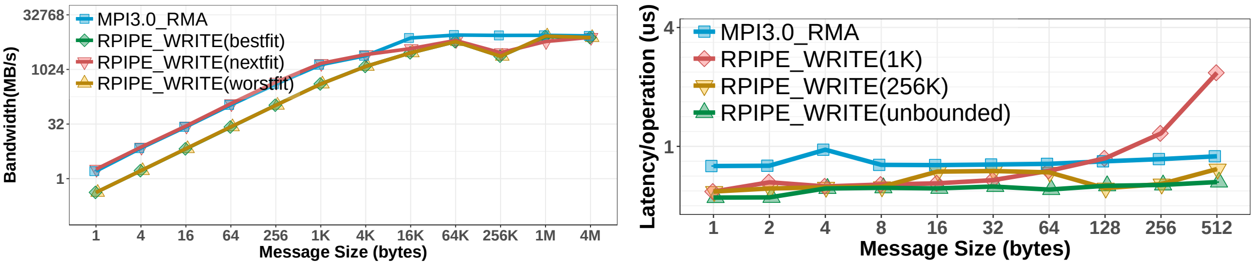
Figure 6.7. **RDMA Bulk Put and Fetch Bandwidth Comparison for MPI and rmalloc in Infiniband^{TM}**

As reported by Figure 6.6a, the `rwrite()` reached peak bandwidth of 8.1GB/s in Cray-XC for bulk large message transfer. Figure 6.6b shows that, `rread()` reached peak bandwidth of 7.9GB/s in Cray-XC for large transfers. This was seen as 13% improvement compared to MPI's peak bandwidth value of 7.0GB/s. The `rread()` throughput reported maximum of 10% difference for small messages than RMA *locks* up to 1K messages, while `rwrite()` reported comparable if not marginally better bandwidth than MPI3.0. Figure 6.7a and Figure 6.7b report results on Infiniband^{TM} when same benchmarks were executed. A peak bandwidth of 1.7GB/s was reached by rmalloc `rwrite()` which was 10% lower than MPI RDMA Put peak bandwidth of 1.9GB/s on Infiniband^{TM} . The peak bandwidth for RDMA GET showed similar trend (See Figure 6.7b). However due to the PSM "backend" optimizations rmalloc in some cases reported upto 2X bandwidth improvement for small to large messages. The rmalloc performance metrics for small and medium-sized messages (up to 4,000) and for larger message latency measurements per operation were comparable to that of MPI-3.0 *locks* and GASNetEx, while the MPI-3.0 RMA barrier/fence was the slowest among the group.

6.1.3. Analyzing Synchronization Cost. Table 6.1 reports a performance comparison for rmalloc when executed with different algorithms for our distributed segregated allocator. We used three message size classes of 4, 512 and 64K as our base payload. The experiments were executed with RDMA PUT and GET in the Cray-XC30 cluster under

	Synchronization Mode				
	Block (next-fit)	Tag (best/worst-fit)		Channel (best/worst-fit)	
Communication Mode	t(us)	t(us)	slowdown	t(us)	slowdown
RWRITE(small,4)	0.687875	1.218790	0.771819	2.412901	2.507761
RWRITE(medium,512)	0.715452	1.275250	0.782440	2.459728	2.438006
RWRITE(large,64K)	10.160331	18.153805	0.786734	36.159626	2.558902
RREAD(small,4)	1.696002	2.981292	0.757835	5.863599	2.457307
RREAD(medium,512)	2.457840	4.347762	0.768936	8.567319	2.485711
RREAD(large,64K)	10.630672	19.135609	0.800038	40.755211	2.504115

Table 6.1. **A Performance comparison for rmalloc synchronization modes (Block, Tag and Channel) for small, medium and large messages in Cray-XC**

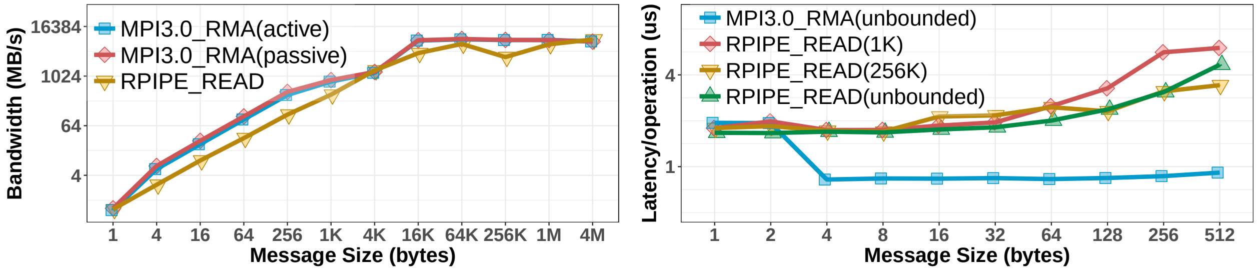


(A) **`rwrite` Bandwidth vs. MPI-3.0**

(B) **Bounded `rwrite` Latency vs. MPI-3.0**

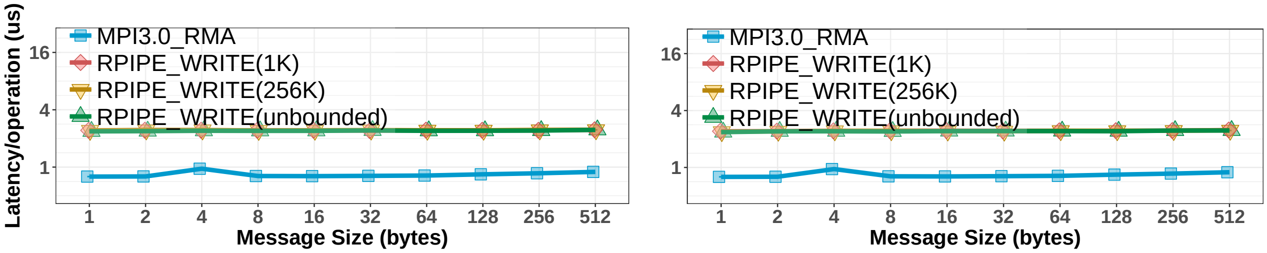
Figure 6.8. **RDMA Put Latency/Bandwidth for MPI, MPI-3.0, and rmalloc**

one configuration for RMALLOC but only switching heuristic algorithm between experiment iterations. As mentioned by Table 3.1 in Chapter 3, each heuristic algorithm is backed by different synchronization protocols. In this scenario we used *block* based protocol for next-fit while both *tag* and *channel* based protocols were utilized for best-fit and worst-fit algorithms. For *tag* protocol executions the RDMA queue length was kept at 25% of the total capacity of the allocator such that trade off between time overhead for synchronization and space for RDMA buffers were kept at an acceptable level. For both RDMA PUT and GET and for all message sizes tested, *channel* based protocol performance showcased the worst. This was expected as the synchronization overhead became higher as synchronization routines



(A) **`rread` Bandwidth vs. MPI-3.0** (B) **Bounded `rread` Latency vs. MPI-3.0**

Figure 6.9. **RDMA Fetch Latency/Bandwidth for MPI, MPI-3.0, and rmalloc**



(A) **Worst-fit** (B) **Best-fit**

Figure 6.10. **Small/Medium Message RDMA Put Latency for MPI-3.0 and RDMA *Managed Buffers* Worst-fit, Best-fit heuristics for 1K, 256K and *unbounded* `rmalloc` allocators**

were executed for each and every allocation request. The amount of slowdown observed was around 2.5X for *channel* synchronization. However, *tag* based protocol reported considerably less overhead around $0.7 - 0.8X$ because execution of synchronization routines were differed until allocators ran out of (RDMA) memory. Our observations did not suggest considerable difference in relative slowdowns for either of the RDMA operations. For all experiments displayed later in this section, we discuss only the best case (i.e. next-fit, with *block* protocol) and the worst case (i.e. best/worst-fit with *channel* protocol) scenarios for performance comparisons.

We compared latency and bandwidth benchmarks for `rwrite()` and `rread()` operations and considered that scenarios differ in synchronization mode and for different allocation heuristics, as well as "bounded" and "unbounded" allocators for "rmalloc". A common usage pattern of "rpipe" is to initialize the "rmalloc" RDMA size to the total transfer length (i.e. *unbounded*, if the total transfer size is known a priori). However, when the total transfer size is not known, such as in the case of "active messages" or total system memory with the *cap* being limited by the applications (i.e. for memory-bound computations), "rmalloc" instance size may need to be regulated. In Figure 6.9, we report latency and bandwidth comparison for RDMA fetch. "rpipe" `rread()` reached peak bandwidth of 7.9GB/s for large transfers and average latency of about 2.14us for small and medium messages. The `rread()` operation performed comparably with MPI-3.0 passive RMA. Its data transfer protocol switches from "FMA GET with AMO" (small messages) to "FMA GET followed by FMA PUT"(medium-sized messages) and finally to "BTE GET followed by FMA PUT"(large messages). The switches happen exactly at message sizes 64-bytes and 1,000 for `rread`. In Figure 6.8a and Figure 6.8b, we report latency and bandwidth measurements of `rwrite()` for RDMA PUT against available "rmalloc" allocation heuristics. "rpipe" `rwrite()` reached the peak bandwidth of 8.1GB/s for large transfers and average latency of about 1.16us for small(0.65us) and medium messages. The inflection points at 4,000 messages is due to the protocol switch between FMA and BTE.

Heuristic next-fit generally performed better than the other versions. This is because in regular scenarios, synchronization is kept at a minimum for next-fit allocation, while having other heuristics using channel synchronization was expensive, due to more synchronization operations. Also in both cases "rpipe" reads and writes perform well for small and medium-sized messages (up to 4,000) and for larger messages latency measurements per operation were comparable to that of MPI-3.0, with active mode (i.e. PSCW or fence) being the slowest among the group. For small messages, `rread()` suffers from larger synchronization overheads when compared to `rwrite()` – unavailability of single synchronized GET (unlike PUT) operation in uGNI causes extra communication work.

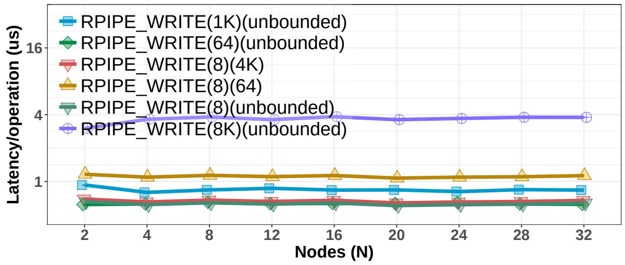
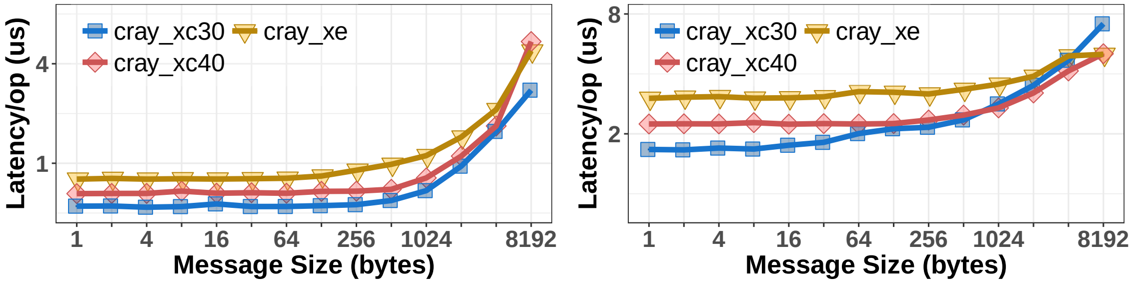


FIGURE 6.11. **Small to Medium Message (8, 64, 1,000, 8,000) Randomized RDMA Write Latency for rmalloc with 64, 4,000, and *unbounded* RDMA channels scaled up to 32 nodes(inter-node)**

In Figure 6.10, we report the performance of `rwrite()` when the allocator size is being regulated for each "rmalloc" allocation mode for `rwrite()` for 1,000, 256,000, and unbounded RDMA transfers. Again, because of the lower number of synchronization operations in *deferred* synchronization w.r.t. *immediate*, allocations based on a `next-fit` (Figure 6.8b) heuristic were faster than `best/worst-fit`(Figure 6.10a and Figure 6.10b). We implemented a multi-threaded latency benchmark for rmalloc with RDMA network operations executed on a selected random node. Simulating a typical parallel application scenario, each rmalloc channel was multiplexed between four network threads and RDMA remote completion (i.e. `rread()`) was tested on a master thread. Figure 6.11 report the `rwrite()` latency when scaled up to 32 nodes, where each node elected a random peer for a PUT operation. We measured latency for message sizes 8, 64, 4,000, and 8,000 and for both "bounded" and "unbounded" allocators for 8-byte messages. Both `rwrite()` and `rread()`[5] performed well at this scale with latency values consistently being in the domain of 0.65us up to 3.8us for the aforementioned message sizes for "bounded" and "unbounded" instances.

6.1.4. Intra-Network Characteristics. The current availability of rmalloc "backend" implementations for Cray platforms provides an excellent opportunity to compare their network characteristics. Cray network fabric feature flavors of HCA's and switches, depending on the type of network topology deployed. The older topology Gemini is a 3D-Torus that

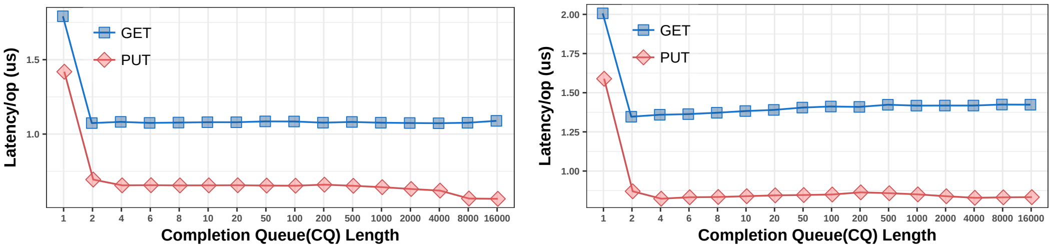
[5] Not reported in plots.



(A) **RDMA PUT**

(B) **RDMA GET**

FIGURE 6.12. **RDMA Latency Comparison with Cray XC30, XC40, and XE**



(A) **Cray XC**

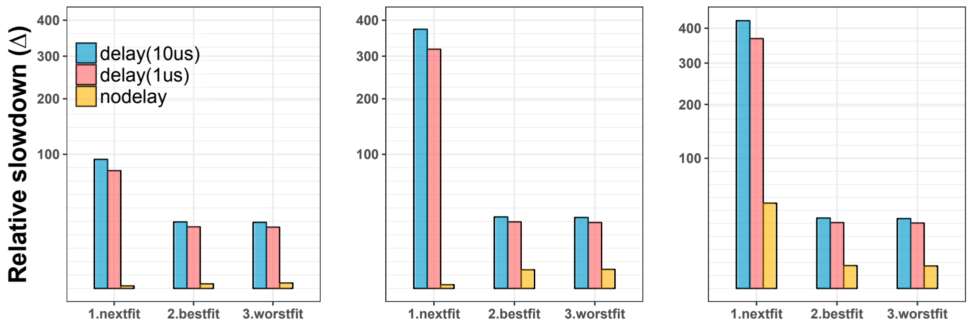
(B) **Cray XE**

FIGURE 6.13. **RDMA PUT/GET performance variance with completion queue length for small messages (4 or 8 bytes)**

comes with Cray XE systems. Cray Aries is a dragonfly topology that that feature XC30 and XC 40 flavors of Cray Systems. Figure 6.12 reports a comparison of RDMA PUT and Fetch latency on small to medium sized messages on XC30, XC40 and XE6. For RDMA Put, RMALLOC reported sub microsecond latency for smaller messages on all platforms, which is an excellent indicator of optimized transport for Cray. However, Cray XC systems reported best case latency, with speedups of up to 15%-30%, compared to Cray XE system. Relatively newer technology faster Cray switch fabric in the Aries network contributed to the aforementioned difference. Similar results were seen with RDMA GET (Figure 6.12b) with Cray XC systems being the more responsive network with fetch latency measurements ranging from 1.6us-3us for smaller messages. The above comparison reported higher latency values for larger message sizes. This is because for these set of experiments transport switch from FMA to BTE was configured for 16,000 messages.

Fine-tuning CQ configuration (See Appendix B) may become an important consideration for RMALLOC usage when number of RDMA channels are scaled. The length of the CQ plays a significant role in the RMALLOC "back end" synchronization algorithm (cf. Chapter 4.5.3), because it will limit the maximum number of in-flight RDMA requests for each RDMA channel. Thus, RMALLOC may exhibit performance degradation with smaller number of completion entries. To test the extent of this performance characteristic, we executed two benchmarks for RDMA PUT and GET, respectively, for varying message sizes. The experiments were similar to a standard latency benchmark except where a number of completion queue entries were increased after each execution. Figure 6.13 reports an instance of latency variation for RDMA PUT and GET for both Cray-XC30 and Cray-XE platforms when CQ size was scaled. The latency drop was significant when the number of CQ events was increased from 1 $\longrightarrow$ 2. However, no significant latency variation was seen for RDMA GET, as CQ events were increased. For RDMA PUT, however, we observed a marginal latency variation of about 10% improvement for the CQ event threshold value set 4000 and above. A similar tendency, where increasing CQ threshold did not result in better latency was evident for medium to larger message sizes as well. Thus, a minimum CQ threshold of 4 for each RDMA channel could be deemed a reasonable assumption when tuning RMALLOC performance in Cray platforms.

6.1.5. Irregular RDMA Access. Evidently, RDMA operations for both "bounded" and best/worst-fit allocation heuristics incur certain overhead, because "rmalloc" allocator need to synchronize and replicate the distributed state between endpoints. Our results suggest that a sufficiently large "rmalloc" instance performs comparably to an unbounded "rpipe". Table 6.2 shows the synchronization overhead and slowdown for `rwrite()`, relative to *unbounded* and the default next-fit allocation when transferring only 8-bytes of data. The overheads increase with the reduction of the allocator size and best-/worst-fit allocations. However, the rate of slowdown implies that a bounded "rmalloc" instance may replace an



(A) **inorder** (B) **reverse** (C) **random**

FIGURE 6.14. **RDMA Irrregular Benchmark `rwrite` vs `rmalloc` heuristics**

mode	t(us)	slowdown	sync_t(us)
RWRITE(4K)	0.663801	1.077	0.023961
RWRITE(64)	1.060588	1.720	0.469532
RWRITE(8)	2.372357	3.847	1.824634
RWRITE(4K,bestfit)	2.464952	3.998	1.798794
RWRITE(64,bestfit)	2.474809	4.014	1.824920
RWRITE(8,bestfit)	2.514217	4.078	1.821569
RWRITE(4K,worstfit)	2.486498	4.033	1.836441
RWRITE(64,worstfit)	2.469965	4.006	1.790187
RWRITE(8,worstfit)	2.484216	4.029	1.807688

TABLE 6.2. **RDMA Pipe Synchronization Overheads for Transferring Data Type uint64_t**

unbounded "rmalloc" to acquire similar performance characteristics. To test the hypothesis that best-fit and worst-fit allocations could outperform the default allocator, we developed a benchmark that performed non-regular RDMA access. In this case, we simulated irregular access by allocating remote memory with "rmalloc", but consuming RDMA buffers out-of-order once messages are completed[6].

We implemented reverse and random (i.e. uniform distribution) ordering, as well as injection of small delays (1us-10us) in the benchmark as shown by Figure 6.14. Contrary to

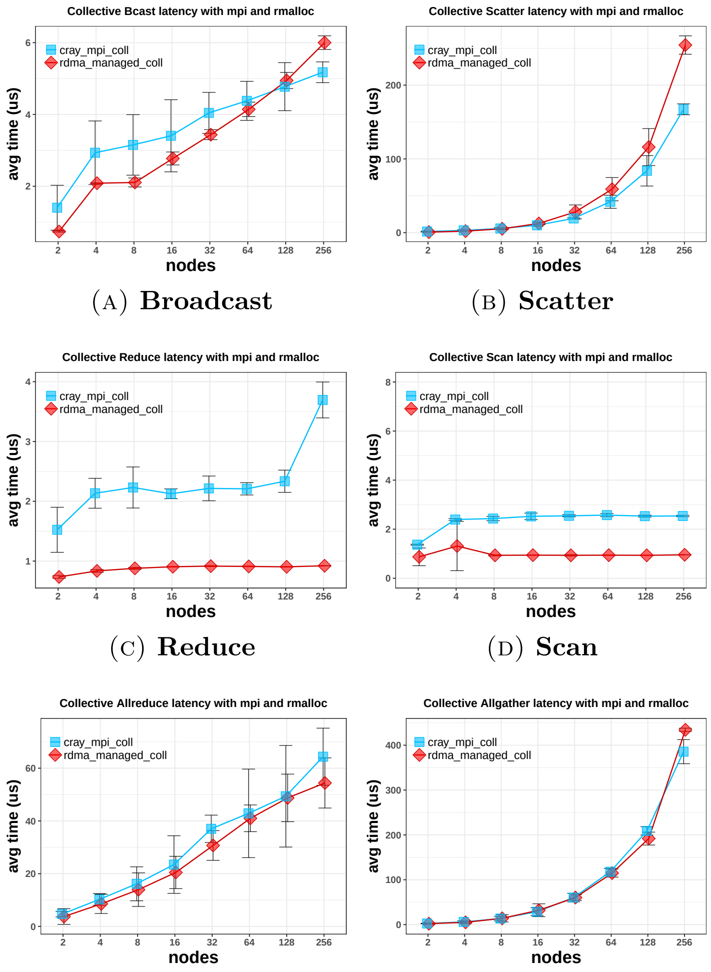
[6] A return from `rread()` call indicates remote completion.

the earlier regular benchmarks, default next-fit allocation reports the worst slowdown compared to best- or worst-fit heuristics (20X-30X compared to 75X to 400X in default). While the difference between all allocators when there is no delay proved marginal, the slowdown increased significantly with the injection of delays as small as 1us. The sub-optimal performance of a next-fit allocator in the irregular case can be attributed to stalls that blow up synchronization delays, even if the number of synchronization operations are small. The next-fit allocator may have enough memory to "commit" into RDMA operations, but the next-fit allocator may wait until all or a subset of out-of-order free requests are coalesced, effectively under-utilizing available memory. For other allocator variants, the distributed allocator releases the *uncommitted* memory as soon as possible – this allows more throughput in an "rpipe" even if the number of synchronization steps increases.

6.2. Collective Performance

Several micro-benchmark experiments were conducted that measured latency for MPI and RMALLOC collective algorithms. We mirrored the collective implementation of the OSU benchmark suite for RMALLOC collectives, such that exact profiling procedures were followed for all experiments. These experiments were executed for both rooted (i.e. Reduce, Broadcast, etc) and non-rooted (i.e. AllReduce, AllGather, etc) collective algorithms on the Cray-XC Bigred II+ cluster up to 256 nodes. Since we wanted to evaluate RDMA performance characteristics, all collective ranks were executed on separate nodes.

As reported by Figure 6.15, in general, the RMALLOC collective latency closely matched or exceeded that of MPI. For all of the non-rooted collectives tested, RMALLOC AllReduce implementation reported 1%-20% speedup compared to MPI while for Allgather this speedup was reduced to about 2%-15%. In certain instances for AllGather (Figure 6.15f), MPI performed better than RMALLOC (3%-10%). For rooted collectives, the most significant results in latency for RMALLOC were observed for the Reduce collective algorithm (Figure 6.15c), with speedup in the range of 51%-75%. Similar speedup for RMALLOC was observed for Scan (Figure 6.15d) with 36%-62%. Both Scan and Reduce algorithms implemented binomial



(A) **Broadcast**

(B) **Scatter**

(C) **Reduce**

(D) **Scan**

(E) **AllReduce**

(F) **AllGather**

FIGURE 6.15. **Collective Latency benchmark results for MPI and `rmalloc` (message size = 8 bytes) in Cray-XC**

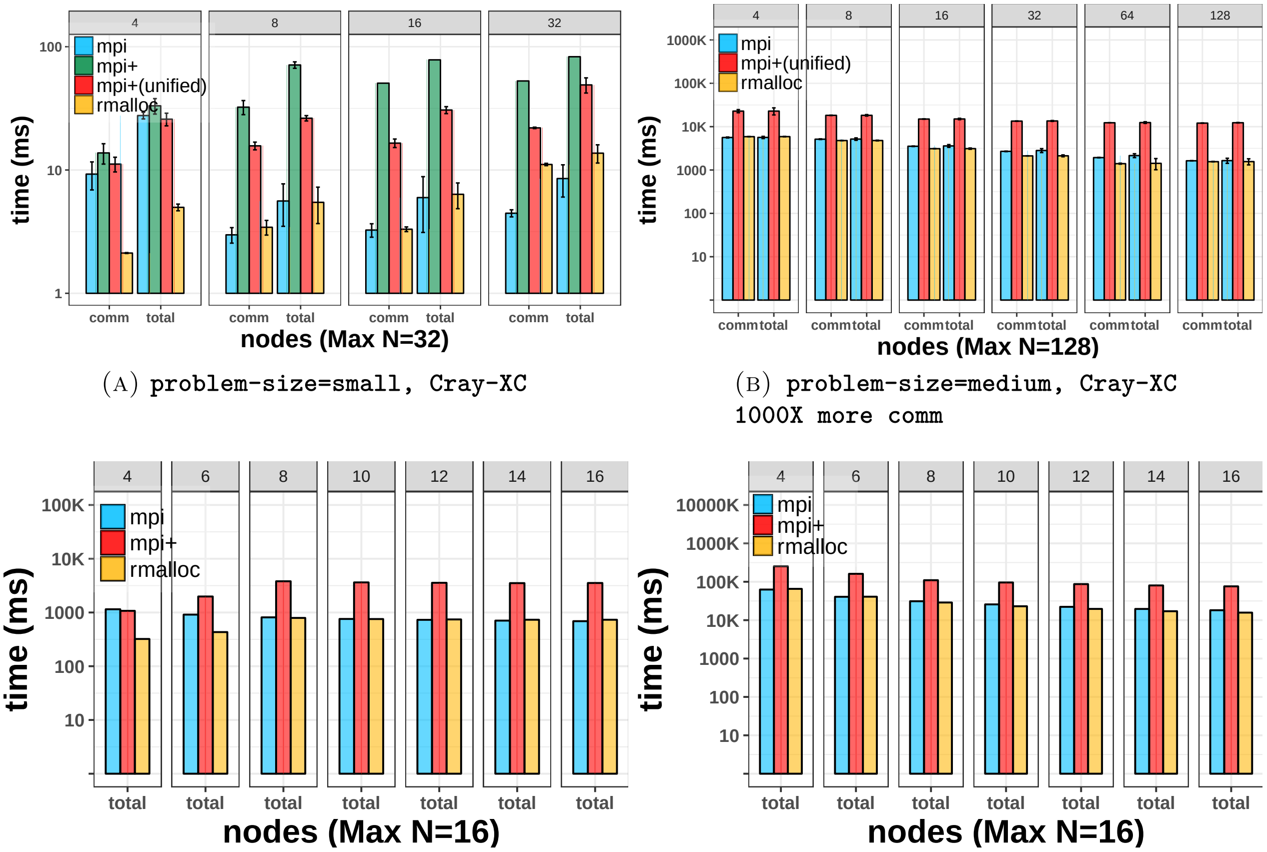
tree topology, which aligned well with the communication overlap supported by underlying RDMA channels. The broadcast algorithm, as shown by Figure 6.15a, reported between 5%-47% improvement in latency measurements w.r.t MPI. However, for $N = 256$, it showed a 15% decline. We purposely implemented a non-scalable, *linear* algorithm for Scatter to compare the performance at different scales. While RMALLOC Scatter reported 6% to 50%

speedup for upto 8 nodes, our observations suggested that the relative advantage of RMALLOC was lost for a larger count of nodes with significant performance degradation of about 25%-52%. It was evident that MPI Scatter had a more efficient version that scaled better.

6.3. Application Performance

We provide three application examples [**119**] for analysis: (1) A stencil exchange pattern to show possible implementation of our RDMA transport for neighborhood exchanges; (2) An irregular (static) graph application called "EM3D" to demonstrate non-uniform data exchange; (3) A distributed hash table that highlights distributed random memory access patterns.

6.3.1. 2D Heat Equation. The neighbourhood relations were fixed for several iterations, such that the three-point stencil (i.e. along the dimension X that was domain decomposed) performs a single Jacobi iteration. At each time step, nodes exchange 'halo' data with their immediate neighbors. A grid point's current temperature depends upon its previous time step value and the values of the neighbouring grid points. The stencil computation may generally follow a traditional bulk synchronous approach of repeated computation and MPI send/recv communication phases. To construct pipelined communication channels with fine-grained completion with RDMA *Managed Buffers* , we modified the stencil program to map halo exchange from each process(i) to its left/right neighbour processes (i.e. $(i-1), (i+1)$) into two rmalloc instances. We set up all remote allocator instances at the initialization phase of the application. Each exchange of the time step inner loop was encoded in `rwrite()` and `rread()` communication and all resolved rmalloc pointers (i.e. pointing to the exchanged *halos*) were directly updated in the outer loop. We selected two particular configurations of communication-heavy stencil exchange where each node exchanged approximately 40,000 and 40 million floating point values during application run time. Figure 6.16 reports a performance comparison for the aforementioned configurations on Cray-XC and Infiniband^{TM} platforms. For Cray-XC experiments the response times (both communication and total time) for smaller communication problems for MPI, MPI3.0 RMA and RDMA *Managed*



(A) `problem-size=small, Cray-XC`

(B) `problem-size=medium, Cray-XC 1000X more comm`

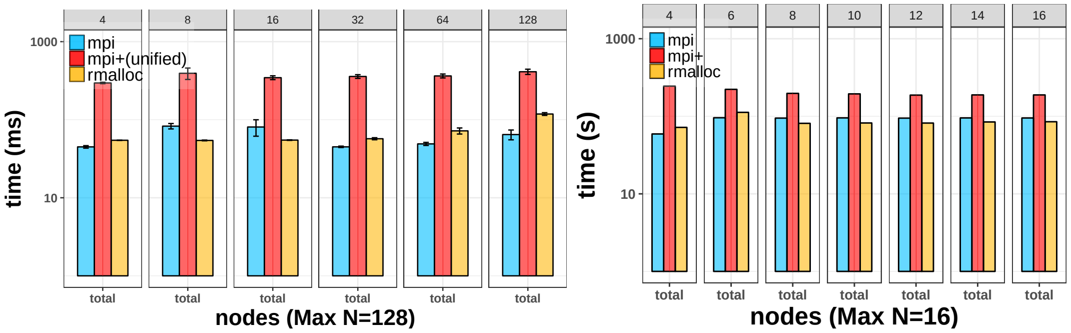
(C) `problem-size=small, Infiniband`$^{\text{TM}}$

(D) `problem-size=medium, Infiniband`$^{\text{TM}}$ `1000X more comm`

Figure 6.16. **Performance results for 2D Heat Equation for MPI,MPI-3.0 and RDMA *Managed Buffers* in Cray-XC and Infiniband$^{\text{TM}}$**

Buffers were measured on up to 32 nodes (Figure 6.16a). Figure 6.16b reports measurements when the problem was scaled up to 128 nodes with 100x more communication operations. The performance measurements for MPI3.0 RMA in Figure 6.16a consisted of response time measurements for implementation flavors of both *unified* and *separate* (i.e. mpi+) execution modes.

For Infiniband$^{\text{TM}}$ experiments (Figure 6.16c and Figure 6.16d) both small and large problems were scaled up to 16 nodes and RDMA *Managed Buffers* 2D heat application was compared against MPI and MPI3.0 RMA (*unified*) implementations. Our measurements suggest that the stencil problem we selected is communication-heavy, thus taking up to 80% of the overall running time. It also verifies that MPI-3.0 RMA performance for *separate* memory mode is understandably slower than its *unified* counterpart. This result is consistent with all of the measurements we collected, therefore, we only reported the best case for MPI-3.0 RMA with *unified* mode in subsequent sections. For both problem sizes, RDMA *Managed Buffers* showed 50%-90% improvement in running time compared to MPI-3.0 RMA, and, was comparable to or better than MPI in certain cases, for example, up to 33% for large problem sizes (Figure 6.16b). In Infiniband^{TM} experiments spanning 16 nodes, RDMA *Managed Buffers* showed an latency improvement of 2%-50% for instances of small problem size and 7%-14% for large problem w.r.t. MPI implementation. MPI implementation reported marginal 1%-6% improvement over RDMA *Managed Buffers* implementation in certain cases for small problem size.



(A) `eM3D, Cray-XC`

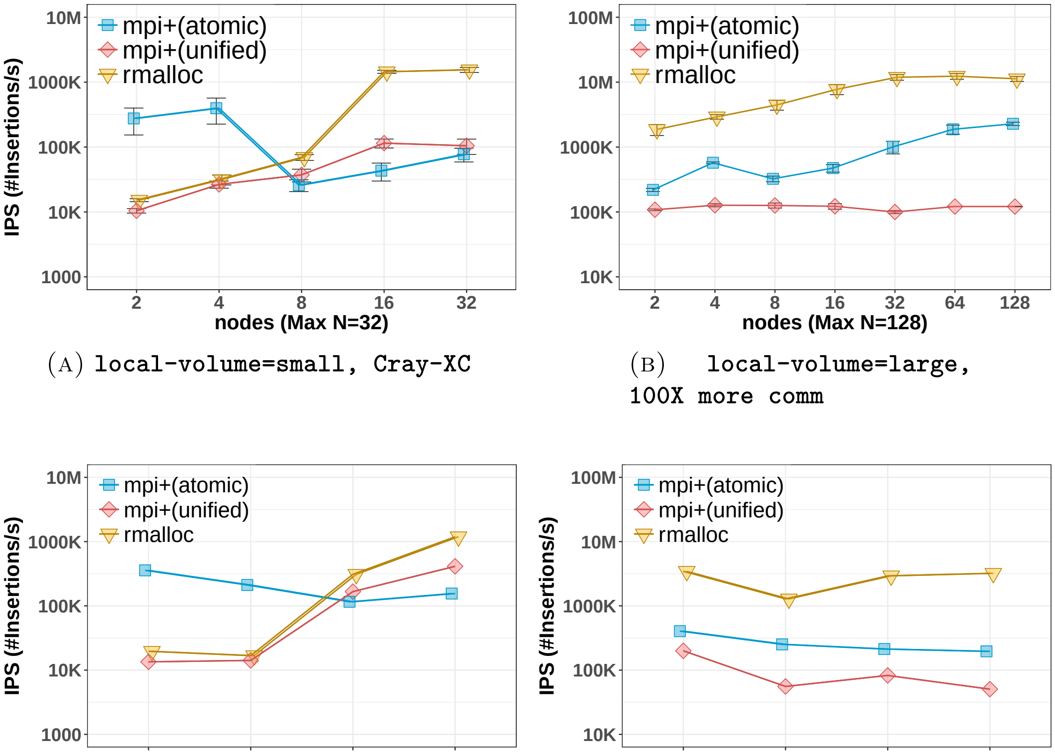
(B) `eM3D, Infiniband`^{TM}

FIGURE 6.17. **Performance results for eM3D application for MPI,MPI-3.0 and RDMA *Managed Buffers* in Cray-XC and Infiniband^{TM}**

6.3.2. EM3D. We described the EM3D parallel algorithm and MP-I3.0 RMA and RDMA *Managed Buffers* implementation variants in subsection 5.2.4. The dependencies between E and H nodes form a bipartite communication graph that is highly relevant for one-sided RDMA communication. Our MPI-1 implementation of EM3D was based on MPI `ISend` and `IRecv` (See Listing 5.4), while MPI-3.0 makes use of `MPI_Win` RMA operations(See Listing 5.7), with locks for fine-grained completion of the communication graph. To eliminate redundant communication edges, certain vertices such as $H3 \longrightarrow E1$ and $H3 \longrightarrow E2$ were proxied by a shared remote allocator instance. At each half-step, the communication routine was initiated for each cell in sub-body by `rwrite()/rread()` but, RDMA *Managed Buffers* only executed synchronization in the $(i-1)^{th} \longrightarrow (i)^{th}$ step transition (See Figure 5.2), in order to fulfill further requests for remote memory. Figure 6.17 shows a performance comparison of EM3D total application running time for a weak-scaling problem up to 128 nodes with an average dependency configuration of $30 \times 6 \times 100$ per node. As reported by Figure 6.17a Cray-XC experiments, for RDMA *Managed Buffers* the EM3D performance gain over the MPI-3.0 RMA (*unified*) version was more than 85% for some cases, and was better or at least comparable[7] against MPI-1 message passing. The application was successfully scaled up to 128 processes with RDMA *Managed Buffers* . To test performance in Infiniband^{TM} platform, EM3D experiments were executed on up to 16 nodes (Figure 6.17b). In general RDMA *Managed Buffers* showed modest performance gains around 30%, although EM3D execution runs in 4 and 6 nodes reported performance degradation of about 15% and 20% over MPI implementation. MPI-3.0 RMA implementation was reported to be the slowest among all instances executed in Infiniband^{TM} .

6.3.3. Distributed Hash Table (DHT). A distributed hash table represents data analytic applications that often require random access in distributed data structures. We adapted the hash-table implementation presented by Gerstenberger et al [**50**] for our transport. We also compared two flavors of MPI-3.0 RMA communication, MPI-3.0 locks and atomic. In this implementation, each process managed a local partition (or volume) of the hash table and an additional heap segment was used to store collisions. The application first searched for an appropriate locality to insert a random element. The MPI-3.0 atomic implementation used atomic fetch/CAS primitives(`MPI_Fetch_and_op`) to test and insert into the first index in the table. If this direct update failed, then a combination of `MPI_GET` and atomic fetch was used to insert a collision to the heap area and then, update relevant pointers of the DHT. The implementation complexity was greatly reduced because remote

[7] MPI-1 was slightly faster (up to 26%) in some cases because it was better optimized for small messages.



(A) `local-volume=small, Cray-XC`

(B) `local-volume=large, 100X more comm`

(C) `local-volume=small, Infiniband`$^{\text{TM}}$

(D) `local-volume=large, 100X more comm`

FIGURE 6.18. **DHT Throughput (Insertions per second) Comparison with MPI-3.0 and RDMA *Managed Buffers* in Cray-XC and Infiniband$^{\text{TM}}$**

memory was able to be directly allocated via `rmalloc()`, while insertion operations were efficiently handled via completion events. Figure 6.18a and Figure 6.18b represent a comparison of insertion rates for small and large communication problems on Cray-XC with respective local volume sizes of 10,000 and 1 million. Figure 6.18c and Figure 6.18d represent a comparison in a Infiniband$^{\text{TM}}$ cluster where experiments were scaled upto 16 nodes. On Cray-XC , RDMA *Managed Buffers* reported a maximum insertion rate of 1.5 and 12.4 million/s on 32 and 128 nodes respectively, significantly outperforming other implementations. Similar results were observed for small and large local volumes for Infiniband$^{\text{TM}}$ with RDMA *Managed Buffers* DHT implementation obtaining maximum insertion rates about

1 and 3.2 million/s respectively. For Infiniband^{TM} experiments as well, RDMA *Managed Buffers* throughput was strikingly better than other implementations when the problem size was scaled with number of nodes. Interestingly on both Cray-XC and Infiniband^{TM} , MPI-3.0 atomic RMA implementation reported better insertion rates (i.e. initial spike) for the small problem size. Evidently, for a smaller number of remote insertions, the probability of collisions was reduced and this resulted in a large number of fast atomic RMA instructions for direct updates. However, this advantage was lost when the number of insertions was increased with volume and number of nodes, and RDMA *Managed Buffers* reported better insertion rates.

CHAPTER 7

Summary and Future Work

"There are two possible outcomes: if the result confirms the hypothesis, then you've made a measurement. If the result is contrary to the hypothesis, then you've made a discovery."

— Enrico Fermi, (1901-1954)

In this work, we introduced RMALLOC , a remote dynamic memory allocator for RDMA enabled high performance computing clusters. RMALLOC is built on top of a network transport, which implements efficient event-driven synchronization protocols in order to minimize overheads in RDMA and achieve similar or better performance to MPI and MPI-3.0 RMA. RMALLOC based RDMA *Managed Buffers* programming abstraction adopts familiar UNIX style interfaces and handles synchronization events in the critical path, transparently and efficiently. Thus, we posit that it is well positioned to benefit many distributed memory parallel applications.

Higher level message passing systems such as MPI are built around multiple layers of abstractions. Empirical evidence suggests that with higher levels of abstraction, larger software overheads are propagated through the runtime. Native RDMA facilities supported by the network fabric separate synchronization from data transfer. This form of decoupling exposes a impedance mismatch between MPI applications, which are either two-sided or too coarse-grained to utilize RDMA efficiently. Additionally, higher level message passing run times avoids direct exposure of a fabric's RDMA capabilities. While this is a reasonable argument for creating a robust set of interfaces, the inability tap into full potential of RDMA fabrics may cause applications to void certain optimizations which are otherwise possible. We showed the existence of high synchronization overhead in MPI for certain situations which attributes to RDMA bootstrap phases, message matching, and ordering. The amount of overhead in a randomized communication benchmark in MPI may range from 0.5% to 40% compared to the native fabric transport.

We presented the advantages of event-driven RDMA for optimizing communication in parallel applications usage where strict ordering and matching constraints for messages are not necessary. Lightweight RDMA events and atomic update capabilities featured by modern network fabrics enable fine grained completion semantics to be implemented efficiently for RDMA synchronization path. Fine-grained synchronization events allowed us to execute optimal zero-copy data transfers without imposing too much overhead in the NICs. RMALLOC implemented communication protocols for syncing the allocator state with the use of network fabric specific hardware assisted synchronization capabilities when possible. It implemented channel, block, and allocation tag level protocols to synchronize dynamic memory allocation instances. Thus, direct memory allocation on remote nodes was made possible for different fragmented state of memory using various heuristic algorithms such as next-fit, worst-fit, random-fit, etc. We also provided a detailed semantic framework to better analyze RMALLOC memory and synchronization interactions for race and deadlock free execution, and also other possible use-cases, such as a tool for compiler assisted transformations of existing parallel programs to RMALLOC. Various usage models of the RDMA managed buffers programming model were presented with relevant examples in both real-world applications and benchmarks.

Our reference implementation proved to be more efficient than respective MPI implementations for the majority of use cases. RMALLOC showed notable performance improvement in standard latency and bandwidth measurements against MPI/MPI-3.0/GASNet and gains in response times for the applications we tested. RMALLOC achieved significant speedups for RDMA PUT or GET network with synchronization flags polling protocol over the traditional method that sent an extra request (up to 3X improvement for small messages). Lightweight on-demand synchronization protocols minimized extra round-trip latency overheads by bulk updates and eliminating waits for local completion. Thus, the aforementioned optimizations were a significant factor towards the reported performance improvements. Current performance evaluations imply that the RMALLOC transport library offers an efficient approach for zero-copy RDMA, within parallel applications. In all scenarios discussed, `rread()` and `rwrite()` operations for RDMA PUT or GET or exceeds the performance of MPI one-sided RDMA in the range of about 2X-8X improvement. The `rread()` and `rwrite()` latency was comparable or marginally improved for low-level transport substrates, such as GASNet.

We analyzed several application classes for testing the use of RMALLOC , as well as new transport techniques in real applications. We showed the basic principle of usage in direct zero-copy message passing in the BSP model, active messages, and communication graphs. In the domain decomposition kernel, communication overhead was decreased up to 90% w.r.t. MPI/MPI-3.0. With RMALLOC, communication graph dependencies of EM3D were programmed with relative ease. The observations from EM3D for large communication graphs indicated that RMALLOC gained up to 85% performance for the number of graph configurations executed. Perhaps, the biggest performance gains were seen in distributed random access kernels. We tested RMALLOC with a distributed hash table implementation with large data sets. While the throughput (i.e. insertions per second) was modest for smaller number of operations, significant throughput gains were observed compared to MPI implementations as the number of communication operations were increased.

RMALLOC , for the most part, showed better usage of RDMA capabilities for modern network fabrics that we tested, thus, they were reflected well by standard performance metrics. However, RMALLOC is not free from certain limitations. While synchronization event capabilities utilized by the library is common in modern RDMA network fabrics, the extent of support rendered by each is highly dependent on the network driver. Certain low-level libraries, such as Cray-uGNI provided hardware support for synchronization events with message size or alignment constraints, or within the constraints of some messaging mode (i.e. FMA). However, older fabric software, such as Intel's PSM (Path Scale Messaging) is optimized for a traditional message passing model. Therefore, it lacked an advanced completion event infrastructure, which limited the implementation of synchronization support in the Infiniband "backend" and reflected relatively poorly in performance. Another important concern was RDMA channel bootstrap overhead [1]) that would increase linearly with the number of point-to-point connections being setup. One mechanism to alleviate channel overheads is to create and retire channels lazily as required via a separate control channel. Furthermore, newer bootstrapping technologies, such as PMIx [**24**], that actively reduce number of bootstrap messages in a clustered system would serve well in this regard. RMALLOC is best utilized as a transport for fast critical path communication in applications or run time systems, such as MPI or PGAS. However, MPI implementations may still suffer from the cost of overheads inherent to the message passing engine, such as message matching and ordering. The opposite scenario was shown true as well, where RMALLOC MPI "back end" based on MPI3.0 RMA, performed significantly poorly w.r.t. the native RDMA "backend" implementations. RMALLOC must address the issue of determining the optimal size for RDMA buffers allocated by each peer to peer channel. The well-known trade-off between space and time will need to be managed effectively in order to make use of current and upcoming extreme-scale systems.

For future work, we plan to extend RMALLOC to other RDMA platforms, such as `RoCE` and Portals. For TCP/IP "backend", the hope is that RMALLOC is able to leverage user space NIC drivers, such as netmap [**99, 60**] and DPDK [**25, 1**], that assist existing transport protocols efficiently. RMALLOC transport library may be extended to other platforms not limited to RDMA, such as GPGPUs. NVlink [**44, 76**] has recently reported great architectural promise as a high throughput PCIe alternative to bridge the gap between CPU and GPU data transfer. Use of RMALLOC in such environments may enable seamless integration of GPU application memory management. The RMALLOC remote dynamic allocator is an extension of the existing operating system memory management library. We believe this work could be a useful insight to linux kernel RDMA infrastructure (i.e. rdma-core) for enabling operating system native remote memory allocation capabilities in the future. It is important to note that I expect that our results will impact application, run time systems, and network library designers by improving existing interfaces and algorithms for best leveraging RDMA facilities

[1] This overhead amounts to the number of requests exchanged for RDMA and memory consumed for channel creation

provided by the hardware. Thus, it is my firm belief that designers should consider low-level optimizations as a major aspect of higher level interface design for maximum utilization of network hardware. We expect that future network libraries and middleware will be designed with those principle in mind.

Bibliography

[1] Intel dpdk. http://www.dpdk.org. 157

[2] Mellanox technologies. programmable connectx-3 pro adapter card. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Programmable_ConnectX-3_Pro_Card_EN.pdf. Accessed: 2019-09-30. 16

[3] Xc series gni and dmapp api user guide. https://pubs.cray.com/content/S-2446/CLE206.0.UP05/xctm-series-gni-and-dmapp-api-user-guide/about-the-xc-series-gni-and-dmapp-api-user-guide. 55, 96, 99, 130, 175

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. 131

[5] S. Agarwal, R. Garg, and N. K. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *International Conference on High-Performance Computing*, pages 280–289. Springer, 2005. 8

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 775–787, 2018. 52

[7] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray xc series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012. 79, 92, 97, 130, 135

[8] 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. 69

[9] B. Barrett, T. Hoefler, J. Dinan, R. Thakur, P. Balaji, B. Gropp, and K. D. Underwood. Remote memory access programming in mpi-3. Technical report, Sandia National Laboratories, 2013. 5, 10, 11, 40, 41, 56, 59, 101, 110

[10] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, Mar. 2008. ISSN 1386-7857. URL http://dx.doi.org/10.1007/s10586-007-0047-2. 8

[11] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *IPDPS 2006.*, pages 10–pp. IEEE, 2006. 3, 53, 55

[12] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*, pages 88–598. IEEE, 2008. 43

[13] R. Belli and T. Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *IPDPS, 2015 IEEE International*, pages 871–881. IEEE, 2015. 5, 6, 52, 56, 59, 95, 96

[14] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015. 13, 74

[15] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015. 176

[16] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel omni-path architecture technology overview. *Intel, Aug*, 2015. 18

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999. 8

[18] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008. 110

[19] D. Bonachea and P. Hargrove. Gasnet specification, v1. 8.1. 2017. 133

[20] J. Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994. 77

[21] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa. Combining message-passing and directives in parallel applications. *SIAM News*, 32(9):10–14, 1999. 8

[22] D. Bryant. Disrupting the data center to create the digital services economy. *Intel Announcement*, 2014. 16

[23] L. P. Carloni, P. Pande, and Y. Xie. Networks-on-chip in emerging interconnect paradigms: Advantages and challenges. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 93–102. IEEE Computer Society, 2009. 1

[24] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt. Pmix: process management for exascale environments. *Parallel Computing*, 79:9–29, 2018. 69, 157

[25] I. Cerrato, M. Annarumma, and F. Risso. Supporting fine-grained network functions through intel dpdk. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 1–6. IEEE, 2014. 52, 157

[26] M. M. Chabbi. *Software Support For Efficient Use of Modern Computer Architectures*. PhD thesis, 2015. 12

[27] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001. 32, 43

[28] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010. 6

[29] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010. 24, 52

[30] G. Chrysos. Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper*, 176, 2014. 43, 46, 48

[31] U. Consortium et al. Upc language specifications v1. 2. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2005. 24

[32] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing'93. Proceedings*, pages 262–273. IEEE, 1993. 118

[33] A. Danalis, A. Brown, L. Pollock, and M. Swany. Introducing gravel: An mpi companion library. In *IPDPS 2008. IEEE*, pages 1–5. IEEE, 2008. 3, 5, 53

[34] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an npb experimental study. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 200–214. Springer, 2005. 24

[35] L. Dickman, G. Lindahl, D. Olson, J. Rubin, and J. Broughton. Pathscale infinipath: A first look. In *13th Symposium on High Performance Interconnects (HOTI'05)*, pages 163–165. IEEE, 2005. 74, 130

[36] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the global arrays pgas model using mpi one-sided communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 739–750. IEEE, 2012. 2, 6, 11, 52

[37] J. Dongarra et al. Mpi: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)*, 2013. 2, 5, 32, 33, 39, 59

[38] N. Drosinos and N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 15. IEEE, 2004. 8

[39] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 283–292. ACM, 2013. 117

[40] S. Ekanayake, J. Cadena, U. Wickramasinghe, and A. Vullikanti. Midas: Multilinear detection at scale. *Journal of Parallel and Distributed Computing*, 2019. 117

[41] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006. 52

[42] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of SC 2008*, pages 19:1–19:12. IEEE Press, Piscataway, NJ, USA, 2008. ISBN 978-1-4244-2835-9. URL `http://dl.acm.org/citation.cfm?id=1413370.1413390`. 8

[43] M. Flajslik, J. Dinan, and K. D. Underwood. Mitigating mpi message matching misery. In *International conference on high performance computing*, pages 281–299. Springer, 2016. 5, 7

[44] D. Foley and J. Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2): 7–17, 2017. 157

[45] P. W. Frey and G. Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009. 12

[46] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid mpi: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 18. ACM, 2013. 44, 45, 46

[47] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: efficient distributed memory programming on multi-core systems. In *ACM SIGPLAN Notices*, volume 48, pages 177–186. ACM, 2013. 44, 46

[48] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004. 34, 132

[49] B. Gerofi, A. Santogidis, D. Martinet, and Y. Ishikawa. Picodriver: fast-path device drivers for multi-kernel operating systems. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 2–13. ACM, 2018. 21

[50] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. *Scientific Programming*, 22(2):75–91, 2014. 5, 52, 54, 59, 96, 151

[51] B. Goglin and S. Moreaud. Knem: A generic and scalable kernel-assisted intra-node mpi communication framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188, 2013. 44

[52] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005. 34, 132

[53] R. E. Grant, M. J. Rashti, P. Balaji, and A. Afsahi. Scalable connectionless rdma over unreliable datagrams. *Parallel Computing*, 48:15–39, 2015. 3, 13, 52

[54] W. Gropp, R. Graham, A. Moody, T. Hoefler, R. Treumann, J. Träff, G. Bosilca, D. Solt, B. de Supinski, R. Thakur, et al. Mpi: A message-passing interface standard version 2.2. In *Message Passing Interface Forum*, 2009. 5, 32, 43

[55] P. Grun. Introduction to infiniband for end users. *White paper, InfiniBand Trade Association*, 2010. 55

[56] 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. 12

[57] D. Grünewald and C. Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, volume 243, 2013. 52, 53

[58] D. Grünewald and C. Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, page 243, 2013. 53

[59] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient collective operations using remote memory operations on via-based clusters. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003. 33, 40

[60] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011. 157

[61] P. H. Hargrove and D. Bonachea. Gasnet-ex performance improvements due to specialization for the cray aries network. In *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pages 23–33. IEEE, 2018. 53, 59

[62] S. Hefty. Rsockets. In *2012 OpenFabris International Workshop, Monterey, CA, USA*, 2012. 5, 52

[63] T. Hoefler and A. Lumsdaine. Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, Aug. 2006. 122

[64] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of SC 2010*, pages 1–11. IEEE Computer Society,

Washington, DC, USA, 2010. ISBN 978-1-4244-7559-9. URL `http://dx.doi.org/10.1109/SC.2010.12`. 8

[65] C. Huang, O. Lawlor, and L. V. Kale. Adaptive mpi. In *International workshop on languages and compilers for parallel computing*, pages 306–322. Springer, 2003. 44

[66] W. Huang, G. Santhanaraman, H.-W. Jin, and D. K. Panda. Design alternatives and performance trade-offs for implementing mpi-2 over infiniband. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 191–199. Springer, 2005. 3, 10

[67] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick. An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect. In *IPDPS, 2014 IEEE 28th International*, pages 1115–1125. IEEE, 2014. 10, 101

[68] L. Ivanov and R. Nunna. Modeling and verification of cache coherence protocols. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196)*, volume 5, pages 129–132. IEEE, 2001. 46, 51

[69] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 184–191. IEEE, 2005. 44

[70] A. E. A. C. B. Jones and J. W. Sanders. Communicating sequential processes. 2005. 33

[71] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016. 12, 13, 17, 18, 21, 52, 53, 66

[72] R. Keller and R. L. Graham. Characteristics of the unexpected message queue of mpi applications. In *European MPI Users' Group Meeting*, pages 179–188. Springer, 2010. 117

[73] E. Kissel and M. Swany. Photon: Remote memory access middleware for high-performance runtime systems. In *IPDPSW 2016*, pages 1736–1743, May 2016. 3, 52, 53, 59, 95

[74] B. C. Kuszmaul. Supermalloc: a super fast multithreaded malloc for 64-bit machines. In *ACM SIGPLAN Notices*, volume 50, pages 41–55. ACM, 2015. 52, 77

[75] D. Lea and W. Gloger. A memory allocator, 1996. 52

[76] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *arXiv preprint arXiv:1903.04611*, 2019. 157

[77] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 16. IEEE, 2004. 34

[78] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 16. IEEE, 2004. 37, 40

[79] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004. 3, 37, 40

[80] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004. 40

[81] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007. 117

[82] P. MacArthur and R. D. Russell. An efficient method for stream semantics over rdma. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 841–851. IEEE, 2014. 5, 52

[83] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991. 49

[84] G. Montry. Openfabrics alliance presentation slides. In *14th Symposium on High Performance Interconnects*, 2006. 12

[85] A. Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009. 32, 43

[86] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker. Automatic mpi to ampi program transformation using photran. In *European Conference on Parallel Processing*, pages 531–539. Springer, 2010. 2, 45

[87] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *International Parallel Processing Symposium*, pages 533–546. Springer, 1999. 53

[88] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998. 24

[89] M. Nussle, M. Scherer, and U. Bruning. A resource optimized remote-memory-access architecture for low-latency communication. In *2009 International Conference on Parallel Processing*, pages 220–227. IEEE, 2009. 18

[90] [Online]. Openfabrics alliance. Available from http://www.openfabrics.org. 12

[91] N. Papadopoulou, L. Oden, and P. Balaji. A performance study of ucx over infiniband. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 345–354. IEEE Press, 2017. 27

[92] M. Pérache, P. Carribault, and H. Jourdren. Mpc-mpi: An mpi implementation reducing the overall memory consumption. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 94–103. Springer, 2009. 2, 44, 45

[93] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001. 102

[94] S. Pickartz, P. Reble, C. Clauss, and S. Lankes. Swift: A transparent and flexible communication layer for pcie-coupled accelerators and (co-) processors. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 371–380. IEEE, 2014. 52

[95] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the cray linux environment core specialization feature to realize mpi asynchronous progress on cray xe systems. In *Proceedings of the Cray User Group Conference*, 2012. 79, 130

[96] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, et al. Why is mpi so slow?: Analyzing the fundamental limits in implementing mpi-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 62. ACM, 2017. 5, 7, 54

[97] C. Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–21. IEEE, 2011. 43

[98] M. J. Rashti and A. Afsahi. Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. In *2008 22nd International Symposium on High Performance Computing Systems and Applications*, pages 95–101. IEEE, 2008. 54

[99] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012. 157

[100] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010. 32, 43

[101] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 40–43. IEEE, 2015. 3, 12, 53, 55

[102] T. Shanley. *Infiniband Network Architecture.* Addison-Wesley Professional, 2003. 102, 176

[103] A. Sodani. Intel xeon phi processor "knights landing" architectural overview. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2015. 16

[104] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016. 43, 46, 48

[105] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson. A ugni-based asynchronous message-driven runtime system for cray supercomputers with gemini interconnect. In *IPDPS 2012 IEEE 26th International*, pages 751–762. IEEE, 2012. 6, 12, 92, 98

[106] H. Tang and T. Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392. ACM, 2001. 2, 44

[107] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 308–314. IEEE, 1998. 37

[108] R. Thakur and W. D. Gropp. Improving the performance of collective operations in mpich. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 257–267. Springer, 2003. 33, 54

[109] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Traff. Investigating high performance rma interfaces for the mpi-3 standard. In *2009 International Conference on Parallel Processing*, pages 293–300. IEEE, 2009. 5

[110] A. Vishnu, P. Gupta, A. R. Mamidala, and D. K. Panda. A software based approach for providing network fault tolerance in clusters with udapl interface: Mpi level design and performance evaluation. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 85. ACM, 2006. 52

[111] A. Vishnu, G. Santhanaraman, W. Huang, H.-W. Jin, and D. K. Panda. Supporting mpi-2 one sided communication on multi-rail infiniband clusters: Design challenges and performance benefits. In *SC*, pages 137–147. Springer, 2005. 2, 10, 42

[112] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of ISCA 1992*, pages 256–266. ACM, New York, NY, USA, 1992. ISBN 0-89791-509-7. 117

[113] U. Wickramasinghe, G. Bronevetsky, A. Lumsdaine, and A. Friedley. Hybrid mpi: A case study on the xeon phi platform. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, page 6. ACM, 2014. 44, 47, 50, 51

[114] U. Wickramasinghe, L. D'Alessandro, A. Lumsdaine, E. Kissel, M. Swany, and R. Newton. Evaluating collectives in networks of multicore/two-level reduction. Technical report, Technical report, Indiana University, School of Informatics and Computing, 2017. 8

[115] U. Wickramasinghe and A. Lumsdaine. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334*, 2016. 33

[116] U. Wickramasinghe and A. Lumsdaine. Characterizing performance of imbalanced collectives on hybrid and task centric runtimes for two-phase reduction. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 129–144. Springer, 2017. 8, 9

[117] U. Wickramasinghe and A. Lumsdaine. Enabling efficient inter-node message passing and remote memory access via a ugni based light-weight network substrate for cray interconnects. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 578–588. IEEE, 2018. 55, 56, 57, 66, 130, 136

[118] U. Wickramasinghe and A. Lumsdaine. rmalloc () and rpipe ()–a ugni-based distributed remote memory allocator and access library for one-sided messaging. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2018. 57, 130, 136

[119] U. Wickramasinghe, A. Lumsdaine, S. Ekanayake, and M. Swany. Rdma managed buffers: A case for accelerating communication bound processes via fine-grained events for zero-copy message passing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 121–130. IEEE, 2019. 57, 148

[120] M. Woodacre, D. Robb, D. Roe, and K. Feind. The sgi® altixtm 3000 global shared-memory architecture. *Silicon Graphics, Inc*, 2005. 44

[121] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe. High performance rdma protocols in hpc. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 76–85. Springer, 2006. 3, 13, 52, 56

[122] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam. I'm not dead yet!: The role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 73–80. ACM, 2019. 1, 52

[123] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi. Nonblocking epochs in mpi one-sided communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 475–486. IEEE Press, 2014. 2, 5, 11, 59

APPENDIX A

Glossary

AM: Active Message 24

AMO: Atomic Memory Operations 86, 97, 135, 141

BTE: Byte Transfer Engine (uGNI) 18

CAS: Compare And Swap Instruction 151

CQ: Completion Queue 3

CTS: Clear To Send Signal 105

DMA: Direct Memory Access 1

FMA: Fast Memory Access (uGNI) 6, 141

HCA: Host Control Adapter 31, 40, 142

IO: Input Output 1

IPC: Inter Process Communication 32

MDH: Meta Data Handle 74

MESI: Modified Exclusive Shared Invalid Cache 51

MMIO: Memory Mapped IO 12

MMU: Memory Management Unit 1

MPI: Message Passing Interface 2, 32

MQ: Message Queue 104

MTU: Maximum Transmission Unit 96, 106

NIC: Network Interface Controller 13, 31, 40

NoC: Network On Chip 1

NUMA: Non Uniform Memory Access 8

OFED: Open Fabrics Enterprise Distribution 23

PGAS: Partitioned Global Address Space 3

PIO: Programmable IO 17, 105

POSIX: Portable Operating System Interface 32, 47, 49

PSCW: Post Start Complete Wait (MPI3.0 RMA) 5, 176

PSM: Performance Scaled Messaging 23, 31

PU: Processing Unit 13

RC: Reliable Connection 102, 103

RDMA: Remote Direct Memory Access 1

RMA: Remote Memory Access 1

RTS: Ready To Send Signal 105

RX: Receive 16, 18

SCIF: Symmetric Communication Interface 47

SDMA: Send DMA (Intel PSM) 18, 105

SMSG: Small Message Transfer (uGNI) 73

SoC: System On Chip 55, 66

TLP: Transport Layer Packet 13

TX: Transmit 16, 18

UD: Unreliable Datagram 27, 103

UUID: A Universally Unique Identifier 74

WQE: Work Queue Element 21

APPENDIX B

RMALLOC API Guide

This section defines the external and internal APIs for RMALLOC to initiate one-sided RDMA functions in either blocking or non-blocking semantics. The external API is the outermost user facing interface that would be suitable for most of the application scenarios for RDMA *Managed Buffers* programming model. The internal APIs are defined for advanced usage and should be utilized with caution. All RMALLOC functions, configuration structures, error codes are defined in `rmalloc.h`.

B.1. External API

This section describe user APIs for creating remote allocators and utilizing them for RDMA operations.

- `int rinit(int *rank, int *total_ranks)` Initialize and executes all bootstrap routines for RMALLOC collectively. Returns `R_SUCCESS` code if initialization was complete. Upon sucessful completion, input pointers *rank*, *total_ranks* will be assigned a logical rank identifier and total logical nodes discovered, respectively.
- `int rexit(void)` Tear-down and cleanup routines for RMALLOC . This operation is invoked at the end of the parallel application and usually is a collective operation. Returns `R_SUCCESS` code if exit was complete.
- `void rpipe(rch_t ch, rmalloc_config_t cfg)` Creates a rpipe RDMA channel with configuration and channel args.
- `raddr_t rmalloc(rch_t ch, uint64_t size)` Allocates memory from a RDMA channel with *size* number of bytes. This operation returns a memory allocation descriptor. It may blocks until a valid allocation (i.e. `ADDR_NULL`) is found.

- `raddr_t rmalloc_nb(rch_t ch, uint64_t size, int *flag)` Same as above operation except RMALLOC won't block if a valid allocation is not available. The test variable $flag$ will return `R_SUCCESS` on successful completion. If completion is pending then the test variable will return `R_PENDING`.
- `void* rmem(rch_t ch, raddr_t d)` Returns a pointer to a virtual memory address for a corresponding RDMA channel descriptor. if descriptor is not a valid allocation (i.e. `ADDR_NULL`) this function will always return a `NULL` pointer.
- `int rwrite(rch_t ch, raddr_t d)` RDMA Put, Atomic Write or Local Write into a virtual address described by the input channel descriptor. This operation will always return after local completion. If operation was successful then it returns `R_SUCCESS` code.
- `int rread(rch_t ch, void *ptr, raddr_t d, int size)` RDMA Get, Atomic Read or Local Read into a virtual address described by the input channel descriptor. As a side effect it will populate size of bytes read into *size*. This operation will always block until completion. If operation was successful then it returns `R_SUCCESS` code.
- `int rread_nb(rch_t ch, void *ptr, raddr_t d, int size, int *flag)` Same as above operation except `rread` will not block if read operation was not complete. The test variable $flag$ will return `R_SUCCESS` on successful completion. If completion is pending then the test variable will return `R_PENDING`.
- `int rread_addr_nb(rch_t ch, void *ptr, raddr_t d, int *flag)` Same as above operation except `rread` assumes that the allocated descriptor is valid (`precondition` : A `rmalloc()` has allocated a valid descriptor). The test variable $flag$ will return `R_SUCCESS` on successful completion. If completion is pending then the test variable will return `R_PENDING`.
- `int rfree(raddr_t desc)` Returns RMALLOC RDMA memory back to the channel. This operation will always return after local completion.

B.2. Internal API

This section briefly describes advanced library specific APIs for utility functions, managing remote allocators and other fine grained protocol operations.

- `uint32_t network_rank(void)` Returns a logical node identifier.
- `uint32_t network_ranks(void)` Returns a total logical nodes available.
- `void network_abort(int e)` Terminates the RMALLOC execution environment forcefully.
- `int network_barrier(void)` Synchronizes all processes in a RMALLOC execution group to a rendezvous point.
- `int network_rdma_send(int to, void *buf, int bytes)` Message passing API to send a direct message to a target node.
- `int network_rdma_recv(int from, void *buf, int bytes)` Message passing API to receive a direct message from a target node.
- `void* network_header(raddr_t desc)` Returns a pointer to a user portion of the message header for a remote/local RMALLOC descriptor. This API could be useful to send immediate data in addition to the primary payload sent to remote buffer.
- `uint16_t network_header_bytes(uint64_t payload_size)` Returns size of a user portion of the message header for an intended payload length.
- `void* getmem(rdma_vmem_t *vmem, raddr_t desc)` Returns a pointer to a virtual memory address for a corresponding RDMA channel descriptor and allocator instance. This returned pointer may point to either physically mapped memory or locally unmapped memory belonging to the remote.
- `int setmem(rdma_vmem_t* vmem, raddr_t desc, void *buf_addr)` Copies data from a local buffer to a RDMA channel descriptor.
- `int rdma_channel_write_async(rch_t ch, raddr_t local, int *flag)` A non-blocking API for RDMA Put or Atomic Write into a virtual address described by the input channel descriptor.

- `void rdma_channel_read_async(rch_t ch, raddr_t recv, int *flag)` A non-blocking API for RDMA Get or Atomic Read into a virtual address described by the input channel descriptor..
- `raddr_t rmalloc_sync(rch_t ch, rdma_vmem_t* vmem, uint64_t bytes)` Allocates memory from a given RDMA local or shadow allocator instance with *bytes* number of total packet size (payload + header). This operation performs implicit network progress and returns a memory allocation descriptor.
- `int rfree_sync(rch_t ch, rdma_vmem_t* vmem, raddr_t desc)` Returns RMALLOC RDMA memory back to the given local or shadow allocator instance. This operation will always return after local completion.
- `void progress_tags(rch_t ch, rdma_vmem_t *shadow, int *test)` Function to ensure progress of all inbound or outbound blocks or tags destined for the instantiated RDMA channel.
- `rch_t rdma_vmem_channel_va(int nargs, ...)` Serves similar function to channel creation (i.e. `rpipe()`) but takes variable arguments as input.
- `rdma_vmem_t* rdma_vmem_create(void* buffer, uint32_t quantum, uint64_t size, int src, int dest, rmalloc_alloc_algorithm_t algpolicy, rmalloc_sync_algorithm_t sync, bool isshadow, bool alloc_with_headers, int flags, rmalloc_config_t *cfg)` Creates an RDMA allocator isntance to manage future allocations and deallocations for a registered RDMA pinned buffers. This function allows binding a particular predefined allocation policy for the local or remote allocator as well as a configuration.
- `int rdma_vmem_destroy(rdma_vmem_t* v)` Cleanup resources for a given allocator instance.
- `raddr_t rdmalloc(rdma_vmem_t* vmem, uint64_t bytes)` Allocates a RDMA buffer locally (i.e. non-shadow) for the requested length of *bytes*.
- `raddr_t rdmafree(rdma_vmem_t* vmem, raddr_t desc)` De-allocates a RDMA buffer locally (i.e. non-shadow) for the given RMALLOC descriptor.
- `rdma_vmem_t* get_rdmalloc_local(rch_t ch)` Returns the local allocator instance corresponding to a given RDMA channel.

- `rdma_vmem_t* get_rdmalloc_remote(rch_t ch)` Returns the shadow allocator instance corresponding to a given RDMA channel.
- `bool channel_active_side(rch_t ch)` Returns a boolen flag indicating whether current logical node acts as RDMA active or passive for the given RDMA channel.

B.3. Configuration API

This section describes API for configuring each RDMA channel. The RMALLOC library contains a separate configuration context for each channel which describes general parameters such as channel width, allocation heuristic and operation type as well as fabric specific parameters such as CQ thresholds.

B.3.1. Channel Parameters.

- `.channels` sets number of primary RDMA channels embedded for this "rpipe".
- `primary_channel.sync` sets primary synchronization algorithm for this channel. Options are based on Network Tag/Block or Channel synchronization – Possible values are, `NET_SYNC_BLKS`, `NET_SYNC_TAG`, `CHANNEL_SYNC` or `NO_SYNC`.
- `sec_channel.enabled` sets enable or disable for secondary synchronization.

B.3.2. uGNI Parameters.

- `.max_cqe` sets the maximum number of uGNI CQ events supported per endpoint.
- `.BTE_transfer` sets message size at which FMA to BTE transfer takes place.
- `.sync_method` sets synchronous operation type for "notified flags with polling" . Supported values are `SYNC_REQ` or `SYNC_W_FLAGS` which refer to synchronous RDMA PUT or an *atomic* update for notifications.
- `.ep_route` sets uGNI endpoint routing mode which are adaptive, dht or basic routing modes [**3**] – Possible values are PERF, `NO_ADAPT`, `NO_HASH`, `NO_RADAPT`.
- `.use_local_comp` sets option for wait for completion or no wait for PUT/GET requests.

B.3.3. MPI Parameters.

- `use_passive_rma` sets option to use either epoch based (i.e. PSCW) or lock based synchronization for MPI network. Setting this true will cause lock based MPI3.0 RMA primitives to be utilized all times.

B.3.4. Infiniband$^{\text{TM}}$ PSM Parameters.

- `.rndv_threshold` sets maximum threshold in bytes for PSM SDMA mode.
- `.max_sys_size` sets Maximum amount of system RDMA buffers (in MB) utilized for unexpected messages.
- `.service_level` sets Infiniband PSM Service Level to be utilized for remote endpoints. Allowed range for this setting is between $0 <= sl < 15$. [**15, 102**]
- `.debug_level` Set the PSM library debug level for diagnostic messages.

B.3.5. rmalloc Channel Parameters.

- `.max_descriptors` sets initial value for maximum number of remote allocator descriptors per RDMA channel.
- `.vm_sync_watermark` sets watermark level as a percentage of the channel capacity for `differed` synchronization algorithm. Possible values for this setting include HALF, FULL or any floating point number between $0 <= l <= 1$
- `.thread_sync` sets a boolean flag to either enable or disable multi-threaded support in RMALLOC (experimental feature).

B.3.6. rmalloc Memory Allocation Parameters.

- `.op` sets RDMA Operation in use. Valid values for this setting include `RDMA_PUT`, `RDMA_GET`, `RDMA_ATOMIC`.
- `.peer` sets the destination rank that acts as either source or target for RDMA operation.
- `.rdma_active` sets whether or not local process acts as the initiator process for the RDMA transaction.

- `.algorithm` sets the heuristic algorithm utilized for RMALLOC remote memory instance.
- `.unit_size` sets the minimum granularity of allocation for RMALLOC remote memory instance. For example *.unit_size* = 4, set aside 4 byte chunks for all future allocations on this channel.
- `.alloc_size` sets an estimate on the maximum payload size (capacity) supported by this instance.
- `.total_required` is only set by RMALLOC library, therefore reserved for library use.
- `.atomic_type` sets the RDMA atomic operation type if op is set to `RDMA_ATOMIC`.

```
rmalloc_config_t cfg = {
  .rmalloc_g_settings = {
    .rmalloc_ch_settings = {
        {
          .channels = 1,
          .primary_channel = {
             .sync = NET_SYNC_BLKS
           }, // primary channel
          .sec_channel = {
             .enabled = false,
             .sync = NO_SYNC,
             .alloc = {
               .alloc_size = 0,
               .algorithm = MAX_ALG,
               .unit_size = 0
             }
          } // secondary channel
        }, /* allocator settings for network sync with blocks*/
        {
          .channels = 1,
          .primary_channel = {
             .sync = NET_SYNC_TAGS
           },
          .sec_channel = {
```

```
        .enabled = false,
        .sync = NO_SYNC,
        .alloc = {
          .alloc_size = 0,
          .algorithm = MAX_ALG,
          .unit_size = 0
        }
      }
    }, /* allocator settings for network sync with tags*/
    {
      .channels = 2,
      .primary_channel = {
        .sync = NET_SYNC
       },
      .sec_channel = {
        .enabled = true,
        .sync = CH_SYNC,
        .alloc = {
          .alloc_size = 1024,
          .algorithm = NEXT_FIT,
          .unit_size = 1
        }
      }
    }, /* allocator settings for channel sync*/
},

/*Cray/uGNI specific settings */
.rmalloc_ugni_settings = {
    .max_cqe = 200, // Max uGNI CQ events supported per endpoint
    .BTE_transfer = 16384, // uGNI BTE (DMA) mode threshold in bytes
    .sync_method = SYNC_W_FLAGS, // remote synchronization method
                              // SYNC_REQ or SYNC_W_FLAGS
    .ep_route = PERF// endpoint routing mode,
                              // PERF, NO_ADAPT, NO_HASH, NO_RADAPT
        .use_local_comp = false, // wait for local completion in network requests
      },

    /*MPI RMA specific settings */
    .rmalloc_mpi_settings = {
        .use_passive_rma = 1, // Option to use active or passive MPI3.0 RMA
      },

    /*Infiniband/PSM specific settings */
    .rmalloc_psm_settings = {
        .rndv_threshold = 8192, // Rendevouz (DMA) mode threshold in bytes
        .max_sys_size = 8 ,// Maximum amount of system buffers in MB for unexpected messages
        .service_level = 1 , // Infiniband SL to use for communication for remote endpoints.
                                // (0 <= SL < 15)
        .debug_level = 0 , // Set the PSM library debug level.
     },

     /* Vmem Instance Settings */
    .rmalloc_vm_settings = {
           .max_descriptors = 1000000, // Max remote allocator descriptors per channel
           .vm_sync_watermark = FULL, // Time to Synchronize vm instance for allocation lists
           .thread_sync = false, // Enable instance level thread safety for multithreaded
    }
  },
  .rmalloc_misc_settings = {
      .use_internal_settings = 1,
          .verbose = 1
  },
  /* general parameters for RDMA channels*/
  .allocator = {
    .op = RDMA_PUT . // RDMA Operation RMDA_* type (PUT, GET, Atomic)
        .peer = −1, // RDMA peer rank
        .rdma_active = false, // RDMA initator side always sets ‘‘true’’
        .algorithm = NEXT_FIT, // Allocation Algorithm
        .unit_size = 1, // Basic allocation unit size
        .alloc_size = 0, // Total payload supported by this channel
        .total_required = 0 // System reserved
        .atomic_type = 0 // Atomic type supported, default = XOR
    },
#ifdef ENABLE_MPI_NETWORK
  .backend = MPI_BACKEND
#endif
#ifdef ENABLE_UGNI_NETWORK
  .backend = UGNI_BACKEND
#endif
#ifdef ENABLE_INFINIBAND_NETWORK
  .backend = INFINIBAND_BACKEND
#endif
};
```

Listing B.1. **A rmalloc configuration example**

B.4. Examples

This section presents two examples of RDMA *Managed Buffers* programs for rmalloc . Listing B.2 reports a simple program for zero copying messages between nodes. Because of the symmetric nature of rmalloc interfaces, the same program can be switched between different RDMA operations. Listing B.3 shows a parallel ring program for exchanging zero copy messages between peers in RDMA *Managed Buffers* program.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

```
#include <string.h>
#include <assert.h>
#include ”basic_cfg.h”

int rank, nprocs;
#define SEND_VAL 1000

#define FOR_EACH_INT(ptr, val, num) \
      do{\
        int _i;\
        for(_i = 0; _i < (num); _i++ )\
                *(ptr+_i) = (val);\
      } while(0)

#define FOR_EACH_ASSERT_INT(ptr, val, num) \
      do{\
        int _i;\
        for(_i = 0; _i < (num); _i++ )\
                assert(*(ptr+_i) == (val));\
      } while(0)

int main(int argc, char *argv[]) {
  rch_t ch;

  bool is_rdma_get = false;
  // Configure channels for either RDMA Get or PUT
  if(argc > 1 && !strcmp(argv[1], ”−get”)){
     is_rdma_get = true;
  } else if(argc > 1 && !strcmp(argv[1], ”−put”)){
     is_rdma_get = false;
  }
  /*Globally Initialize rmalloc*/
  rinit(&rank, &nprocs);
```

```
int  peer = (rank + 1) % nprocs;
size_t unit = 4;
/* Allocation size is 4 bytes */
size_t allocsz = sizeof(int);
size_t  chunks = allocsz/sizeof(int);

/* Initialize RDMA Channel size*/
int CH_SIZE = allocsz * 20;
assert(nprocs == 2);
int is_source = !rank;
int N = 1;

cfg.allocator.peer = peer;
cfg.allocator.unit_size = unit;
cfg.allocator.alloc_size = CH_SIZE;
cfg.allocator.op = RDMA_PUT;
cfg.allocator.algorithm = BEST_FIT;
if(is_rdma_get){
  cfg.allocator.op = RDMA_GET;
}
if(cfg.allocator.op == RDMA_PUT){
  cfg.allocator.rdma_active = is_source;
} else {
  cfg.allocator.rdma_active = !is_source;
}
// Channel bootstrap
rpipe(ch, cfg);
raddr_t addr ;
int *ptr;
int i;
if (is_source) {
    /*RDMA zero copy sender */
    for (i = 0; i < N; ++i) {
      addr = rmalloc(ch, allocsz);
      ptr = rmem(ch, addr);
      FOR_EACH_INT(ptr, SEND_VAL * (i+1), chunks);
      rwrite(ch, addr);
    }
  } else {
    // RDMA receiver
    for (i = 0; i < N; ++i) {
      rread(ch, addr, ptr, allocsz);
      FOR_EACH_ASSERT_INT(ptr, SEND_VAL * (i+1), chunks);
      rfree(ch, addr);
    }
  }
  printf(”finalized␣rmalloc␣program␣for␣rank␣:␣%d\n”, rank);

  rexit();
  return 0;
}
```

Listing B.2. **A symmetric RDMA *Managed Buffers* program to send zero copy messages via either RDMA PUT or GET**

```
#include <assert.h>
#include <rmalloc.h>
#include <stdbool.h>
```

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include ”basic_config.h”

int rank, nprocs;
#define SEND_VAL1 1000

void channel_write(rch_t ch, int val, size_t msg_sz) {
  raddr_t addr = rmalloc(ch, msg_sz);
  int * r = rmem(ch, addr);
  *r = val;
  rwrite(ch, addr);
}

bool channel_test(rch_t ch, int test, size_t msg_sz) {
  raddr_t addr;
  int * r;
  bool ret;
  rread(ch, addr, r, msg_sz);
  ret = (*r == test);
  rfree(ch, addr);

  return ret;
}

int exec_ring() {
  // Select right and left ranks in the ring
  int right = (rank + 1) % nprocs;
  int left = (rank − 1) < 0 ? nprocs − 1 : (rank − 1);
  size_t msg_sz = sizeof(int);
  size_t channel_sz;
```

```
int i, n = 128;
int N = n + 512;

// Select channel width
channel_sz = n * (msg_sz);
int num_channels = 2;
if (!network_rank()) {
  printf(
      ”\nstarting␣ring␣channel␣test␣=>␣channels␣:␣[total=%d,␣per␣rank=%d]”
      ”\nmsg␣:␣[%lu␣bytes]”
      ”␣channel␣:␣[%lu␣bytes]␣channel␣width␣:␣[channel_sz/msg_sz␣=␣%d]”
      ”␣total␣width␣:␣[%d]\n␣”,
      num_channels * nprocs, 2, msg_sz, channel_sz, n, N);
}

// create channels for RDMA send/recv
// each rank has a Rmalloc send and recv channels
rmalloc_config_t *cfgs = malloc(sizeof(rmalloc_config_t) * num_channels);
rch_t * ch = malloc(sizeof(rch_t) * num_channels);

// config rmalloc send−>recv channels
for (i = 0; i < num_channels; ++i) {
  cfgs[i]    = cfg;
  cfgs[i].op  = RDMA_PUT;
  cfgs[i].allocator.unit_size = msg_sz;
  cfgs[i].allocator.alloc_size = channel_sz;
  if (i % 2 == 0) {
    // for even indices set left RDMA passive
    cfgs[i].allocator.peer = left;
    cfgs[i].allocator.rdma_active = 0;
  } else {
    // for odd indices set right RDMA active
    cfgs[i].allocator.peer = right;
```

```
      cfgs[i].allocator.rdma_active = 1;
    }
  }

  // we have to be careful in a ring of rmalloc pipes
  // On the first processor A pipe for 'right' direction
  // has to be created first to avoid deadlock!!
  if(!rank){
    rpipe(ch[1], cfgs[1]);
    rpipe(ch[0], cfgs[0]);
  } else {
    rpipe(ch[0], cfgs[0]);
    rpipe(ch[1], cfgs[1]);
  }

  // do comm
  for (i = 0; i < N; ++i) {
    // write via rmalloc ch
    channel_write(ch[1], SEND_VAL1 + i, msg_sz);
    TEST_ASSERT(channel_test(ch[0], SEND_VAL1 + i, msg_sz));
  }
  return TEST_OK;
}

int main(int argc, char *argv[]) {
  rinit(&rank, &nprocs);
  exec_ring();
  rexit();
  return TEST_SUITE_END();
}
```

Listing B.3. **A parallel program to stream zero copy messages via a RDMA ring overlay.**

Curriculum Vitae – Udayanga S. Wickramasinghe

Research Interests

Computer Systems (Operating Systems, Networks, Parallel/Concurrent Systems), High Performance Computing, Computer Architecture

Education

Indiana University, Bloomington, IN, USA

Ph.D., Computer Science GPA: 3.95/4.0, Jan 2020

- Thesis Topic: *A Hardware and Software Approach for Optimizing Communication with Direct Memory Access Techniques*
- Advisor: Andrew Lumsdaine, Ph.D

M.S., Computer Science, Dec 2016

- GPA: 3.89/4.0

University of Moratuwa, WP, Sri Lanka

B.S., Computer Science and Engineering, Oct 2010

- GPA: 3.83/4.2 (First Class Hons, Rank: 5th/100)

Professional Experience

Associate Instructor Aug 2017 to Present

Department of Computer Science,
Indiana University.
Course: Advanced Operating Systems(P536)
Projects:

- **rmalloc** https://github.com/uswick/rmalloc (2017−2018) A high performance Dynamic RDMA memory allocator. "rmalloc" also has a thin RDMA access layer called "rpipe" that expose zero-copy network (i.e. read/write) operations. – with Andrew Lumsdaine (Ph.D)
- **QInspect/Autograder/MossCheck** (2017−2018) Set of tools for partially validating XINU/Linux operating system commands on QEMU environment or host OS. Mosscheck is a plagurism checking tool built on Stanford MOSS tool. – with Andrew Lukefahr (Ph.D), Bryce Himebaugh
- **Mobydick** (2017−2018) Co-Developed boot-loader(x86 64bit long mode) and bootstrap memory manager module for a Multi–Exo Kernel operating system – aims to support shared nothing paradigm on many core systems as an end-to-end philosophy of OS design. – with Watshala Vithanage

Summer Intern May 2019 to July 2019

VMware (VMware Monitor (Hypervisor) Group) - Palo Alto,
Supervisor(s): Vinay Kumar, Fred Jacobs, Peter Snyder
Projects:

- **vHVFuzz** - A Fuzzing tool for Virtual Machine Monitor(VMM) which targets in-guest utilize of hardware assisted virtualization (vHV). vHVFuzz use nested virtualization interface to attack VMM by fuzzing Intel-VT/AMD virtual machine context structures. vHV Fuzz use multiple fuzzing strategies which include randomized fuzzing guided by rules and more informed choice fuzzing with AFL (code coverage)

Research Assistant Aug 2013 to May 2017

CREST (Center for Research Extreme Scale Technologies),
Indiana University.
Supervisor: Andrew Lumsdaine, Ph.D
Projects:

- **Photon** (2016−2017) https://github.iu.edu/OPEN/photon A High performance RDMA library. Developed a collective libPhotonNBC comm library on top of a flexible notification event abstraction (i.e. PUT/GET with completion) – with Martin Swany (Ph.D), Ezra Kissel (Ph.D), Andrew Lumsdaine (Ph.D)
- **HPX-5** (2015−2017) Contributed to scheduler and parallel group communication runtime of HPX-5 AMT reference implementation for ParelleX specification. – with Luke D'Allessandro (Ph.D), Ezra Kissel (Ph.D), Andrew Lumsdaine (Ph.D) and Martin Swany (Ph.D)
- **Xinu for Zynq/Parallella Platform** (2015−2016) Ported Xinu kernel for upcoming embedded platform Parallella baseboard - Implemented ARM corex-A9 drivers for UART, interrupt and timers in Xilinx Zynq-7000 PS for Xinu. – with Andrew Lumsdaine (Ph.D) and Martin Swany (Ph.D)
- **Flow/Sight** https://github.com/uswick/sight (2014−2015) A continuous/streaming query engine prototype for key/value data streams and a distributed log analysis tool – with Greg Bronevetsky (Ph.D), Andrew Lumsdaine (Ph.D)
- **Hybrid MPI** (2013−2015) A MPI framework optimized for intra-node message passing communication for modern multicore and many-core architectures. Ported HybridMPI for Xeon-Phi co-processor – with Andrew Friedley (Ph.D/Intel), Greg Bronevetsky (Ph.D/GoogleX), Andrew Lumsdaine (Ph.D)

Summer Intern May 2017 to Aug 2017

Pacific Northwest National Laboratory (PNNL),
Supervisor: Andrew Lumsdaine, Ph.D
Projects:

- Characterizing Irregularity in Reductions for Asynchronous Many Task Runtimes

Student Intern May 2015 to Aug 2015

Lawrence Livermore National Laboratory (LLNL),
Supervisor: David Bohme, Ph.D
Projects:

- **Caliper** – a light weight context annotation system for profiling, performance monitoring and logging. – with David Bohme (Ph.D), Martin Schulz (Ph.D)

Research Assistant Aug 2012 to May 2013

Department of Computer Science,
Indiana University Purdue University Indianapolis
Projects:

- Structural analysis of "monomer" proteins using spectral clustering methods. – with Jing Yuan Liu (Ph.D)
- Server Administrator (Linux) and Network management for data servers of large scale bio-informatics data.

Senior Software Engineer (WSO2 Inc.) Sep 2010 to June 2012

Products:

- WSO2 ESB (Enterprise Service Bus)
- WSO2 Solution Architecture

Apache (ASF) PMC Member and Committer May 2011 to Present

Projects:

- Apache Synapse

Teaching Experience

Associate instructor Spring 2018

P436/536 - Introduction to Operating Systems/Advanced Operating Systems
Instructor: Martin Swany, Ph.D
Department of Computer Science,
Indiana University

Associate instructor Fall 2017

P536 - Advanced Operating Systems
Instructor: Andrew Lukefahr, Ph.D
Department of Computer Science,
Indiana University

Google Summer of Code Mentor Summer 2012,2013

Apache Synapse Project
Google Inc.

Journal Publications

1. Saliya Ekanayake, Jose Cadena, **Udayanga Wickramasinghe**, Anil Kumar Vullikanti "MIDAS: Multilinear Detection at Scale." IEEE
International Journal of Parallel and Distributed Computing *JPDC*, 2018(Accepted).

Conference, Workshop Publications

1. **Udayanga Wickramasinghe**, Andrew Lumsdaine, Saliya Ekanayake, Martin Swany “RDMA Managed Buffers: A Case for Accelerating Communication-Bound Processes via Fine-Grained Events for Zero-Copy Message Passing” IEEE International Symposium on Parallel and Distributed Computing *ISPDC*, 2019.

2. **Udayanga Wickramasinghe**, Andrew Lumsdaine “rmalloc() and rpipe()-a uGNI-based Distributed Remote Memory Allocator and Access Library for One-sided Messaging” *HPDC*(High-Performance Parallel and Distributed Computing) Workshop on Runtime and Operating Systems for Supercomputers *ROSS*, 2018.

3. **Udayanga Wickramasinghe**, Andrew Lumsdaine “Enabling Efficient Inter-node Message Passing and Remote Memory Access via a uGNI based Light-weight Network Substrate for Cray Interconnects” *CCGrid* (Cluster, Cloud and Grid Computing) Workshop on Advances in High-Performance Algorithms Middleware and Applications *AHPAMA*, 2018.

4. Saliya Ekanayake, Jose Cadena, **Udayanga Wickramasinghe**, Anil Kumar Vullikanti “MIDAS: Multilinear Detection at Scale.” IEEE International Parallel and Distributed Processing Symposium *IPDPS*, 2018.

5. **Udayanga Wickramasinghe**, Andrew Lumsdaine “Characterizing Performance of Imbalanced Collectives on Hybrid and Task Centric Runtimes for Two-Phase Reduction” Languages and Compilers for Parallel Computing: 30th International Workshop *LCPC*, 2017.

6. **Udayanga Wickramasinghe**, Andrew Lumsdaine “A Survey of Methods for Collective Communication Optimization and Tuning” CoRR abs/1611.06334 (arXiv 2016)

7. **Udayanga Wickramasinghe**, Greg Bronevetsky, Andrew Lumsdaine, Andrew Friedley, Andrew Lumsdaine “Hybrid MPI: a case study on the Xeon Phi platform.” *ICS* (International Conference on Supercomputing) Workshop on Runtime and Operating Systems for Supercomputers *ROSS*, 2014.

8. **Udayanga Wickramasinghe**, Charith D. Wickramarachchi, Pradeep R. Fernando, Dulanjanie Sumanasena, Srinath Perera and Sanjiva Weerawarana “BISSA: Empowering Web gadget communication with tuple spaces” Workshop on Gateway Computing Environments *GCE*, 2010

Technical Reports, Abstracts, Posters

1. **Udayanga Wickramasinghe**, Vinay Kumar, Fred Jacobs, Peter Snyder "vHVFuzz: A Blueprint for Discovering Software Vulnerabilities in Nested Virtualization" 2019.

2. **Udayanga Wickramasinghe**, Luke D'Alessandro, Andrew Lumsdaine, Ezra Kissel, Martin Swany, Ryan Newton "Evaluating Collectives in Networks of Multicore/Two-level Reduction" 2017.

3. **Udayanga Wickramasinghe**, Ezra Kissel, Andrew Lumsdaine "LibPhotonNBC: An RDMA Aware Collective Library on Photon" 2016.

4. **Udayanga Wickramasinghe**, David Bohme, Todd Gamblin and Martin Schulz "Supporting High Level Types and Reduction Functions for Performance Data Collection", LLNL Student Poster Symposium, 2015.

5. **Udayanga Wickramasinghe** "Towards a Scalable Communication Model with Hybrid MPI and OpenMP on Xeon Phi", 2014.

6. Pradeep R. Fernando, **Udayanga Wickramasinghe**, Charith D. Wickramarachchi, Dulanjanie Sumanasena, Srinath Perera and Sanjiva Weerawarana
"BISSA: Empowering Web-Gadget Communication with Tuple Spaces" 8th International Workshop on Middleware for Grids, Clouds and e-Science *MGC*, 2010

Peer Reviewed Articles

- IEEE Transactions on Big Data (Journal) July 2019

Awards

Graduate Travel Awards

- Supercomputing Conference, Denver, CO Nov 2013
- Supercomputing Conference, New Orleans, LA Nov 2014
- PNNL Travel grant for Workshop on Languages and Compilers (LCPC) June 2017
- IU Travel grant for Cloud and Grid Computing Conference (CCGrid) May 2018
- Travel grant for Symposium on High-Performance Parallel and Distributed Computing (HPDC) July 2018

Undergraduate Awards — University of Moratuwa

- Dean's List - 4 semesters (awarded for achieving SGPA above 3.80) 2006–2010
- Google Summer of Code (Apache Xerces project) May 2010

College Awards

- Dialog Telekom Fellowship 2002–2003
  - Outstanding achievement (top 25) in Advanced Ordinary Level Island-wide
    Examination, Srilanka 2003

Presentations

Conference/Workshop Talks

- AHPAMA, Washington DC May 2018
- LCPC , College Station, TX June 2017
- ROSS, Munich, Germany June 2014

Indiana University

- CREST All-Personnel Technical Talk - Hybrid MPI May 2014

Hardware and Software Skills

Computer Programming:

- *compiled* – C, C++, Java, Assembly(x86/arm) – *functional* – Scheme – *scripting* – python, Ruby, R, JavaScript, shell/bash/tcl/tk, perl – *build systems* – GNU make, Autoconf/Automake, Cmake, Maven, Ant – *declarative* – Microsoft SQL, MySQL and others