# Overlapping Communication with Computation in MPI Applications

Valeria Cardellini
Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Rome, Italy
cardellini@ing.uniroma2.it

Alessandro Fanfarillo
Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Rome, Italy
fanfarillo@ing.uniroma2.it

Salvatore Filippone
School of Aerospace, Transport and Manufacturing
Cranfield University, Cranfield, UK
salvatore.filippone@cranfield.ac.uk

Università di Roma "Tor Vergata"
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Technical Report RR-16.09

February 2016

**Abstract**

In High Performance Computing (HPC), minimizing communication overhead is one of the most important goals in order to get high performance. This is more than ever important on exascale platforms, where there will be a much higher degree of parallelism compared to petascale platforms, resulting in increased communication overhead with considerable impact on application execution time and energy expenses. A good strategy for containing this overhead is to hide communication costs by overlapping them with computation.

Despite the increasing interest in achieving computation/communication overlapping, details about the reasons that prevent it from succeeding are not easy to find, leading to confusion and poor application optimization.

The Message Passing Interface (MPI) library, a de-facto standard in the HPC world, has always provided non-blocking communication routines able, in theory, to achieve communication/computation overlapping. Unfortunately, several factors related with the MPI independent progress and offload capability of the underlying network, make this overlap hard do achieve. With the introduction of one-sided communication routines, providing high quality MPI implementations, able to progress communication independently, is becoming as important as providing low latency and high bandwidth communication.

In this paper, we gather the most significant contributions about computation/communication overlapping and provide technical explanation of how such overlap can be achieved on modern supercomputers.

# 1 Introduction

Overlapping computation with communication have the potential to improve the performance of parallel applications. This is more than ever important in the exascale era [5], where the high degree of parallelism will require asynchronous data transfers in order to scale on billions of cores.

Implementing computation/communication overlapping schemes was already possible in the first MPI standard, where non-blocking routines like *MPI_ISend* and *MPI_IRecv* give the feeling that the actual communication is happening in the "background", while the main program is busy doing number crunching work. This fortunate situation can be obtained only when the underlying network interconnect and the MPI implementation work in tandem to achieve this goal. Unfortunately, this is quite hard to get.

With the introduction of one-sided communication routines in MPI-2 and their refinement in MPI-3, overlapping communication with computation became even more tempting because of their almost perfect match with the Remote Direct Memory Access (RDMA) capabilities exposed by several, widely used, interconnects like Infiniband [18], RDMA over Converged Ethernet (RoCE) [3], Cray Gemini [2] and Aries [9].

Currently, the one-sided semantics defined in the standard MPI-3 allows Partitioned Global Address Space (PGAS) languages to implement their runtime library on top of MPI-3 [10, 15, 8]. One of the most important features of PGAS languages is the ability of performing direct gets or puts into a remote processor's memory (target process), (supposedly) without any interaction on the remote side. It is obvious that achieving an high level of overlapping in such cases is very important [30].

Because of the hardware dependency and low-level details involved in obtaining overlapping, MPI courses and tutorials usually focus more on instructing people about "how to get correct results and hoping to get overlap" rather than explaining how it actually works. Based on our experience, it is quite common to find people involved in HPC that are not able to explain why a certain code does not show any sign of overlapping, even though the MPI design would allow that.

With the introduction of the passive MPI one-sided routines, users get even more confused about asynchronous data transfers. In fact, such routines look like a perfect match with RDMA operations exposed by modern network interconnects, giving the idea that the hardware will take care of progressing the communication asynchronously while the main program is busy with other tasks.

In this work, we gather the most important contributions in literature about computation/-communication overlapping in MPI and provide concrete suggestions and explanations about why and how it is possible (or not possible) to get computation/communication overlapping on modern supercomputers.

The paper is organized as follows: in Section 2 we introduce the reader to MPI and to the issues related with message progress. Most of the material explained in this section has been taken from [11]. In Section 3 we describe RDMA and how the MPI one-sided routines map this technology. In Section 4 we define what overlap and progress mean and how the lack of one of them prevents asynchronous communication. In Section 5 we describe how MPI implementations use different protocols in order to provide the best performance possible, and how these protocols impact the ability of overlapping communication with computation. In Section 6 we describe the Remote Memory Access (RMA) characteristics of the Cray Aries interconnect, the APIs exposed, and how to get overlapping using the Cray MPI implementation as well as with alternative MPI implementations. In Section 7 we provide our experience in overlapping communication with computation using the most used MPI implementation supporting Infiniband: MVAPICH2 [22]. We also describe the most effective designs able to provide truly passive MPI one-sided routines using

Infiniband atomic operations. Finally, in Section 8 we provide our conclusions and future trends.

## 2 Introduction to MPI

The High Performance Computing (HPC) world has been dominated for years by the parallel distributed memory paradigm, which allows to distribute the computation among several compute units, each of which has a private memory space. Such approach has been successfully used in the last 30 years, when multi-core architectures were unavailable. Programming a distributed memory system implies the usage of a mechanism to allow interprocess communication. In the early 80s, the dominant approach for inter-process communication on distributed systems was the message passing model. At that time, every machine/network producer provided its own message passing protocol to its users. In 1992, a group of researchers from academia and industry, decided to design a standardized and portable message-passing system called MPI, Message Passing Interface. Since then, MPI has been the de-facto standard for communication among processes on distributed memory systems and thus, the dominant programming system in HPC.

The MPI standard documents have grown and evolved over time. After MPI-2.0 was released, the effort of the MPI community has been devoted more on improving and featuring MPI implementations than evolving the standard. Some MPI implementation projects focused more on portability, providing implementations able to run on several platforms, networks, and operative systems. Others, following customers' requests, provided MPI implementations for specific networks like Myrinet [27], InfiniBand [18] and Quadrics [23].

The MPI specification provides over 200 API functions with a specific and defined behavior to which an implementation must conform.

Although the specification is standard, there are many cases where text is ambiguous and the MPI implementer has to make a decision. The asynchronous message progress, which is the main topic of this work, is one of them.

The main reason that guides the implementer is usually performance. A communication layer provides high performance when it provides low latency and high bandwidth during data transfers. It is obvious that characteristics such as latency and bandwidth are strictly related to the network layer. Thus, an MPI implementation can span over multiple layers of the OSI stack. Depending on the specific network capability exposed by the network layer, an MPI implementation may span from simple byte transfer to complex routing tasks.

In the HPC world, several networks provides MPI-like interfaces in order to support MPI functions in hardware. Portals, Quadrics' Tports interface, Myrinet's MX interface and Infinipath's PSM interface are a few of them. These networks provide an interface very similar to the user-level MPI functions. In most cases, the network hardware supports abstractions similar to MPI's core concepts such as process identification, unit message transmission and tag matching. As a consequence, an MPI implementation on top of these interfaces is quite minimal and can provide high performance because of the hardware-based implementation. However, even with such MPI friendly networks, it is not possible to optimize same-node communication using shared memory. For example, fusing shared memory and network hardware MPI message matching, typically introduces an high overhead that negates any speed advantage that hardware matching alone could provide. For this reason, MPI implementations relying on these MPI-like interfaces have to implement MPI message matching in software. Other networks interfaces and transport layers like TCP sockets, Ethernet frames, OpenFabrics verbs and shared memory, provide abstraction models that do not exactly match MPI's requirements. In this case, an MPI implementation must translate MPI concepts, like message matching and progress, to the underlying network.

# 3 RDMA and One-Sided MPI Communication

Network communication is usually conducted through several memory copies to/from user-space and kernel-space, supervised by the CPU. Such process is very expensive in terms of CPU and it gets worse with the increase of network speeds. The problem is that, while CPU is busy managing communication tasks, it cannot be used for other work, such as actual user-level computation.

A common approach used in electronics and computer science, is to offload the entire task, or just a part of it, to specialized hardware. Although offloading relieves CPU from the burden of managing communication, it has the typical drawback associated with offloading technologies: getting data to and from the accelerating hardware. The commonly used approach is Direct Memory Access (DMA): allowing a device to have direct access to memory without involving the CPU.

The natural extension of DMA to networks is called RDMA (Remote Direct Memory Access). RDMA enables efficient access to memory of a remote host, as well as processing offload in order to provide high performance. A very powerful characteristic of RDMA is the possibility, for the sender, to move data to a memory region of the remote process without waiting for a receive call. In other words, the remote process does not actively participate in the communication process because everything is implemented in hardware. Such paradigm is usually called one-sided, because only one entity is aware of communication.

The RDMA protocols lie right below the application layer; it includes high level protocol for exchanging RDMA messages and also the protocols needed for placing the data (zero-copy and OS-bypass). At the transport layer, RDMA can either use a specialized transport layer, like Infiniband, or can utilize an existing one like TCP (iWARP).

RDMA provides two kind of semantics: message and memory semantics. The first is also known as Send/Receive and requires that both, sender and receiver, are aware of the data transfer. In fact, the sender must gather up all the part of a message to do a single send, whereas the receiver must commit to receiving the message. The second semantic is the closest to the RDMA concept, where the sender moves data directly to the remote memory region without requiring any receive call from the receiver.

In general, RDMA is composed by three main components: zero-copy, OS-bypass and protocol offload. Zero-copy is the capability to send and receive data without making copies. Such capability is quite difficult to enable on the receive side; in fact, RDMA operations must contain sufficient information to tell the remote side where to put data.

OS-bypass refers to the capability for the user application, to have direct access to the hardware network without involving the kernel. Data buffers must be registered in advance in order to inform the hardware adapter of the buffer's physical location. Another requirement for the memory buffers is that they must be "pinned", in order to prevent the OS to swap-out that portion of memory and make it unreachable.

RDMA requires also protocol offloading. It means moving RDMA and transport layer protocol processing (or a part of it) to specialized hardware on the NIC. How much of the protocol processing (more generally the network interface) is implemented on the NIC varies from almost all to almost none; the trade-off is not easy to decide. Proponents of onload justify this choice by pointing to the high performance of CPUs; in fact, the low performance of embedded processors installed on NICs can easily become a bottleneck when the number of offloaded network capabilities increases. The QLogic (Intel True Scale) network adapters are an example of this category.

On the other hand, proponents of offload argue that the ability to overlap communication and computation can be easily provided by offloading operations to the NIC. Infiniband adapters designed by Mellanox belong to this category.

The MPI-2 standard provides the programmer with one-sided communication routines, these

functions allow one to issue get and put operations on remote memory. Compared to the usual MPI two-sided approach (point-to-point), the new one-sided approach shows great potential, in particular for those applications that get benefits from computation/ communication overlapping. The MPI one-sided routines map naturally on RDMA; however, the synchronization modes of the MPI-2 one-sided communication pose a few issues.

MPI-2 provides two kinds of synchronization modes: active and passive. In active mode, the target is actively involved in synchronization; it is required to invoke a common RMA barrier or to define an exposure RMA epoch. In [19] Jiang et al. show how to implement a RDMA based design for MPI-2 active one-sided communication using the Infiniband capabilities.

On the other hand, in passive mode, the target is not explicitly involved in synchronization. In other words, with the passive mode, the one-sided operations have to be performed at the target node independently of what the target process is doing. This is very similar to the memory semantics provided by RDMA and how one-sided communication should work.

Anyway, the main challenge is to provide a mechanism to make independent *progress* of passive MPI one-sided communications. In [20], Jiang et al. propose four different design alternatives to implement efficient MPI-2 passive one-sided operations using Infiniband capabilities.

In order to understand the difficulties of providing such mechanism (even on RDMA enabled networks), we need to describe more precisely what *overlap* and *progress* mean.

# 4 Overlap and MPI Progress

In this section we give a definition of *Progress* and *Overlap*, and their impact on the performance of MPI applications; this section is based on [6].

*Overlap* is a characteristic of the network layer; it consists of the NIC capability to take care of data transfer(s) without direct involvement of the host processor, thus allowing the CPU to be dedicated to computation.

*Progress* is a characteristic related to MPI, which is the software stack that resides above the network layer. The MPI standard defines a Progress Rule for asynchronous communication operations; unfortunately, there are two different interpretations of this rule leading to different behaviors, both compliant with the standard.

The strictest interpretation of the Progress Rule is that once a non-blocking communication operation has been posted, the subsequent posting of a matching operation will allow the original one to make progress regardless of whether the application makes any further library calls. In short, this interpretation mandates non-local progress semantics for all non-blocking communication operations once they have been enabled.

The weak interpretation allows a compliant implementation to require the application (on the receiver/target process) to make further library calls in order to achieve progress on other pending communication operations. At first glance, this choice looks quite shallow; as if it were implemented only to hounour the Progress Rule. Anyway, it should be noted that asynchronous progress is much more effective for large messages than for short messages, mainly because of the higher latency imposed by a thread-safe MPI implementation and polling/interrupt costs, needed by an helper thread-based approach. Applications that do not provide opportunities for overlapping, certainly prefer an MPI implementation which provides lower latency than asynchronous progress.

In general, it is possible to support overlap without supporting independent MPI progress. For example, an InfiniBand network subsystem can usually perform RDMA operations, thus fully overlapping communication and computation, at transport level. Unfortunately, Infiniband receives messages in FIFO order and it does not provide any capability for message selection (required by

MPI two-sided routines). For this reason, the MPI progress engine has to be invoked in order to progress MPI messages; this usually happens when the MPI library is invoked. If the transfer to/from the target address depends on the user application making an MPI library call, then progress is not independent from the computation.

Conversely, it is also possible to have independent MPI progress without overlap.

Asynchronous message progress is a very intricate and controversial topic in high-performance computing [6, 16, 17]. With the current available high-performance networks, there are essentially three strategies for making progress: manual progress, thread-based progress, and communication offload.

The manual progress gives complete control and responsibility to the programmer for implementing message progress. The programmer has to explicitly invoke functions like *MPI_Test* or *MPI_Probe*/*MPI_Iprobe* inside the code in order to progress the communication. Although this solution increases code complexity, it is quite used in several cases because the MPI implementation is much more faster without the burden of the full asynchronous support. Unfortunately, it is very unlikely that the user invokes a progress function like *MPI_Test* or *MPI_Probe*/*MPI_Iprobe* at the right moment. It will probably come too early, where there is nothing to progress, or too late, wasting the opportunity to overlap.

The thread-based progress has been often considered the most effective because enables fully asynchronous progress without any user interaction. There are essentially two possible models for implementing the thread-based progress: 1) polling based; 2) interrupt-based. In the first model, a communication thread is dedicated to each MPI process in order to handle incoming messages from other processes. Each thread polls the MPI progress engine and gets the data immediately. On the other hand, the interrupt-based approach uses hardware interrupts to wake up a thread and make progress exactly at the right time. Even though this approach does not waste resources by continuously polling the network, it suffers of substantial overhead introduced by the OS intervention.

Offloading the protocol processing to the communication hardware (hardware offload), is a very powerful alternative to ensure fully asynchronous progress. Anyway, delegating more work to the communication hardware (into the NIC) can easily become a performance bottleneck due to the lower performance of the embedded processors compared to regular CPUs.

In [17], Hoefler et al. describe and analyze the thread-based approach. They conclude that the thread-based progress based on polling (by-passing the OS) is beneficial only when separate computation cores are available for the progression thread. Using an interrupt-based approach (passing through the operating system) might be helpful in the case of oversubscribed nodes (the progress and user threads share the same core). Anyway, passing through the operative system raises two concerns: 1) it seems unclear how big the interrupt latency and overheads are on a modern systems; 2) the scheduler has to schedule the progress thread right after the interrupts arrives to achieve asynchronous progress. This second issue can be faced by using real-time functionalities in the Linux kernel.

Si et al. [28], recently proposed to use dedicated communication processes (called ghost processes) for ensuring message progress, and employ the MPI-3 shared memory capability for transferring the data from the ghost process to the target process.

# 5 MPI Protocols

The communication cost (Tcomm) of sending data through a network interconnect can be represented by the following equation:

$$Tcomm = Latency + MsgSize \times Bandwidth$$

Besides the message size, it is influenced also by two characteristics of the network interconnect: latency and bandwidth. Latency is a function of the location of sender and receiver on the network and the time spent by the operative system, at the receiver side, to recognize that a message arrived. It dominates the cost of transferring small messages. Bandwidth is a function of network bandwidth and network traffic. It dominates the cost of transferring big messages.

MPI implementations typically use different algorithms (or protocols) for sending messages between peers. In order to provide the best performance, the protocol selection is based on message size. The way these algorithms are implemented, and when they are used, depends on the MPI implementation; anyway, every MPI implementation usually relies on (at least) two protocols: eager and rendezvous. A general description of these two protocols is given in Section 5.1. A detailed description of the eager and rendezvous protocol implemented in Cray MPI is described in Section 6.3.1.

Eager and rendezvous protocols are usually employed for implementing the two-sided MPI functions where both, sender and receiver, are aware of communication. For the MPI one-sided functions, message size is theoretically not influent; what matters is the definition of the *exposure epoch*[1] (synchronization messages). The messages required for defining such epoch and for the actual data transfer may be issued immediately, implementing and *eager* protocol, or delayed until the closure of the exposure epoch (*lazy* protocol). We present an example of two approaches for passive one-sided transfers in section 5.2.

## 5.1 Eager and Rendezvous Protocols

For the classic MPI two-sided paradigm, in order to provide good performance, the MPI implementations use two protocols for message delivery: eager and rendezvous.

In the eager protocol, the sender assumes that the receiver has enough memory space to store the message; this allows to send the entire message immediately, without any agreement between peers. This algorithm is the best approach possibile for achieving high performance: it has minimal startup overhead and provides low latency for small messages. Of course, when the message size grows over a certain threshold, the assumption of enough space on the receiver side does not hold anymore. In this case, some kind of "handshaking" between the processes involved in the transfer must be implemented. This protocol is called rendezvous and, although less efficient than the eager protocol, allows one to transfer big messages with minimal impact on performance.

Implementing these MPI protocols using the right network methods, at the right moment, is critical for getting high performance. A concrete example of how these protocols are implemented in modern networks is described in Section 6.3.1.

## 5.2 Eager and Lazy passive MPI One-Sided Routines

In Section 4, we defined overlap as a network capability. Such definition, although correct, generates confusion among programmers when passive MPI one-sided routines are used.

In order to obtain computation/communication overlapping at application level, programmers usually write something like Code 1, hoping that the *MPI_Put* will progress asynchronously while the CPU is performing the compute intensive part.

[1]For a more detailed explanation we recommend reading [13, 14].

Code 1: Sender overlap

```
MPI_Win_lock(lock_type, dest, 0, win);
MPI_Put(a, 100, MPI_INT, dest, offset, 100, MPI_INT, win);
// Lot of computation here
compute_intensive_kernel();
MPI_Win_unlock(dest, win);
```

Assuming to have a NIC capable of performing overlap, and an MPI implementation that provides independent progress, the MPI standard guarantees that all the RMA operations have been completed at origin and target when *MPI_Win_unlock* returns.

As for the non-blocking MPI two-sided routines, the standard does not tell anything about "when" the communication will happen, but only that it will be completed after a certain point in the code. Also in this case, there are two possible approaches: *eager* and *lazy*. The descriptions reported in this section have been taken from [31].

The eager approach is what almost any programmer would expect: the *MPI_Win_lock* blocks until the lock is granted by the target process, the *MPI_Put* is issued to the network and the transfer is progressed asynchronously during the computation. Finally, the *MPI_Win_unlock* function (possibly) waits for the transfer to complete and releases the remote lock. Such approach allows one to overlap computation with communication but requires two synchronizations (and three messages): one for the lock request/grant and one for the unlock.

On the other hand, with the lazy approach, the origin process does nothing but enqueues the "lock" and "put" requests without blocking. During the *MPI_Win_unlock* the origin process issues the lock request, waits for the grant from the target process, issues the "put" request waiting for its termination and releases the lock. Such approach does not allow any overlap because all the operations are executed at the end of the RMA epoch. Compared to the eager protocol, this approach requires less synchronization because the last operation coincides with the lock release. Furthermore, operations directed towards the same target can be grouped together, resulting in a lower synchronization overhead.

# 6 Cray Aries Interconnect

The Cray Aries interconnect is currently the most advanced network interconnect delivered along with the Cray XC supercomputers. It provides hardware capability for true asynchronous data transfer, and thus represents a good candidate for showing how the RMA capabilities provided by the network can be exploited by the MPI implementations.

In this section, we gather the architectural characteristics of the Aries interconnect provided by [1] in order to provide a solid background for understanding the challenges to computation/-communication overlapping. We also provide hints on how to exploit this interconnect, in order to improve the performance of applications based on one-sided communication.

All the tests have been run on Edison and Cori: two Cray XC30 installed at NERSC.

## 6.1 Aries RMA

Cray Aries provides two remote memory access methods: Fast Memory Access (FMA) and Block Transfer Engine (BTE). FMA is an onload protocol, fully managed by the CPU; as already stated in Section 3, onload protocols provide very low latency because of the high processing speed provided by the CPU. When using FMA, user processes generate network transactions consisting of Puts,

Gets and AMOs (atomic operations) by writing directly to an FMA window within the NIC. The FMA unit translates I/O writes by the processor into fully qualified network requests. FMA supports scattered accesses by allowing the user library to select which bits of the offset into the FMA window determine the remote address and which the remote process. Such characteristics make FMA well suited for small, scattered, message transfers. On the other hand, because the CPU is fully involved in the transfer, it does not allow asynchronous transfers at network level.

FMA allows four types of transactions:

- Short Message (SMSG): each connection uses a dedicated buffer allocated on the remote peer called *Mailbox*. The maximum size of SMSG message is 64 KB.
- Shared Message Queue (MSGQ): uses the facilities provided by SMSG but allows all job instances on a node to share the message buffer resources required by SMSG connection.
- FMA Distributed Memory (FMA DM): it is used for moving user data between local and remote memory. An application prepares a Transaction Request Descriptor, which has properties such as type (PUT/GET), CQ, data source and destination, and length. To post the transaction, an application passes the pointer to a Transaction Request Descriptor to the *PostFma* function.
- Atomic operations (AMOs): it is based on Transaction Request Descriptors, as for FMA DM, but uses different fields.

SMSG and MSGQ provide a reliable messaging protocol with send/receive semantics. Both are implemented using an RDMA Write operation with remote notification. FMA DM is used to execute RDMA Write, RDMA Read and AMOs. These two mechanisms provided by FMA match the two semantics (message and memory) provided by RDMA described in Section 3.

The BTE is an offload protocol, mostly managed by the ASIC processor installed on the Aries NIC. Even though more time is required to set up the data transfer, the BTE method does not require any further involvement by the CPU. An application can direct the Block Transfer Engine (BTE) to perform an RDMA PUT operation, which instructs the BTE to move data from local to a remote memory, or an RDMA GET operation, which instructs the BTE to move data from remote to local memory and to notify a source and destination upon completion. These functions write block transfer descriptors to a queue in the NIC, and the BTE services the requests asynchronously. Block transfer descriptors use privileged state, so access to the BTE is gated through the kernel. Due to the overhead of accessing the BTE through the operating system, the BTE mechanism is more appropriate for larger messages. A *PostRdma* function posts a descriptor to the RDMA queue for BTE to execute a data transaction. The descriptor contains all information about the transaction such as destination process, memory address, handler, etc.

### 6.1.1 Completion Queue Management

Aries (and Gemini) *Completion Queues* (CQ) provide a light-weight notification mechanism to determine when:

- the data in a RDMA Write or non-fetching atomic transaction has been delivered to the target;
- the result data for RDMA Read or fetching atomic transaction has been delivered into local memory;

- a message from a remote peer has been delivered to a previously registered local memory region.

In general, a "TX" CQ is used for the first two types of notifications and separate "RX" CQ is used for the third type. An application, like MPI, can bind a CQ to a registered memory region to receive notifications for messages arriving in that memory region. The *CqCreate()* function creates a CQ, which can be linked to an end point, or a region of memory for incoming messages. The application can check for presence of *Completion Queue Events* (CQEs) on a CQ by polling or waiting for an event (blocking). The function used for checking the CQ is *CqGetEvent()*. The CQE includes application specific data, information about the type of transaction and the status.

## 6.2 Cray Aries/Gemini Software Stack

In order to allow system software to realize as much of the hardware performance of the Gemini and Aries network ASIC as possible, Cray provides two sets of API's: uGNI (user level Generic Network Interface) [24] and DMAPP (Distributed Memory APPlications) [29].

uGNI exposes FMA and BTE control; it is well suited for runtime implementations of one-sided and two-sided communication paradigms. By default, Cray MPI uses a MPICH2 distribution based on a Nemesis driver for Aries/Gemini, layered on uGNI. DMAPP supports Partitioned Global Address Space languages, using one-sided communication and providing collective operations. DMAPP hides the FMA and BTE support and selects the best mechanism based on message size. Cray MPI provides a DMAPP-based version which provides excellent overlap for inter-node one-sided communication. We will provide more details about Cray MPI and how it uses the capability of the Cray interconnect in Section 6.3.

Figure 1 (taken from [1]) shows clearly the software stack organization and the OS-bypass capability required by RDMA.

## 6.3 Cray MPI based on uGNI

In this section, we describe how the default Cray MPI implementation, distributed along with Cray supercomputers, uses the uGNI layer. Cray MPI does not distribute its source code publicly, thus a deep analysis of how the RMA methods are used is not possible; anyway, we will try to describe how the network capabilities are used, based on observations gathered during the experiments and/or taken from Cray documentation. As shown in Figure 1, Cray MPI provides also an implementation based on DMAPP.

### 6.3.1 Overlap in MPI Two-Sided

For MPI two-sided routines, Cray MPI based on uGNI employs four main pathways: 2 eager and 2 rendezvous. The pathway selection is based on message size; the default thresholds are shown in Figure 2. It is easy to guess that small messages will be most likely sent using the FMA method, whereas big messages will use BTE. Although MPI protocols (eager and rendezvous) may look like a perfect match with the network methods (FMA and BTE), their implementation using such methods makes computation/communication overlap difficult to achieve. The following descriptions of the four paths are taken from [24].

**E0 - SMSG Mailbox** The E0 pathway is based on a messaging facility called GNI Short Message (SMSG); such facility is completely based on the FMA Short Message transaction. This mechanism provides the highest performance in terms of latency and short messages rates but requires an

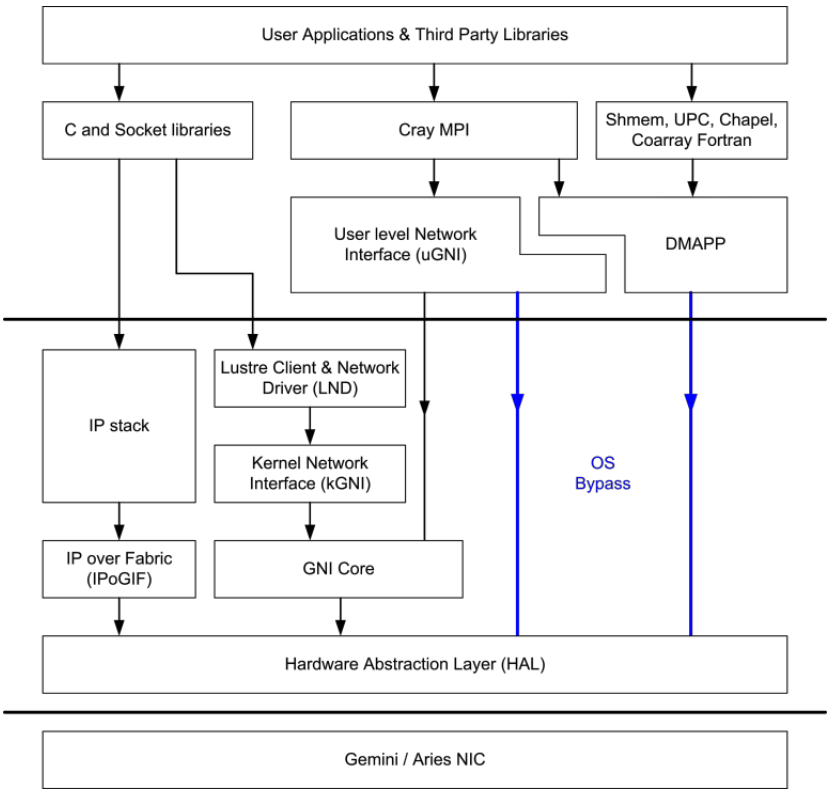


Figure 1: Cray network software stack

amount of memory which grows linearly with the number of connections. If two processes want to communicate, each of them must allocate memory space for a dedicated mailbox. The mailbox size is strongly influenced by the number of processes in the job (Figure: 3). As shown in Figure 2, the E0 protocol is used for messages with size up to 512 bytes (by default).

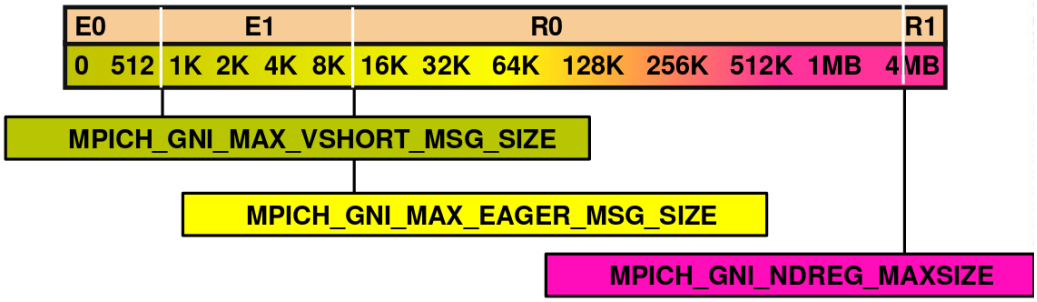


Figure 2: Pathway threshold

The sender knows exactly where to put the data because the mailbox on the receiver side has been already allocated; it also knows that there will be enough space on the receiver side to store the entire message. When the transfer is completed, the receiver gets an event notification in the "RX" CQ which includes both user data and transaction status information. Such event is taken from the queue during a MPI_Recv or MPI_IRecv on the receiver side.

This scheme allows a very effective computation/communication overlap when non-blocking routines are used. In fact, the sender will never block because it knows that on the receiver side a

| Ranks in Job | Max user data (MPT 5.3 ) | MPT 5.4 and later |
|---|---|---|
| < = 512 ranks | 984 bytes | 8152 bytes |
| > 512 and <= 1024 | 984 bytes | 2008 bytes |
| > 1024 and < 16384 | 472 bytes | 472 bytes |
| > 16384 ranks | 216 bytes | 216 bytes |

Figure 3: SMSG Mailbox size

buffer, big enough to contain the data, is already available. The receiver will transfer the data from the buffer to the right user memory segment with a simple and fast *memcpy*. Anyway, because the CPU is involved in the transfer (FMA), this approach is suitable only for small messages.

**E1 - RDMA Get** If the message is larger than the SMSG mailbox, an RDMA read path is used. The sender allocates a DMA buffer and copies the message data into the buffer. A small control message is sent through the SMSG channel to the receiver. The message contains informations necessary for the receiver to be able to do a RDMA Get of the message from the senders's buffer. A simple ACK protocol is then used in order (for the sender) to know that the receiver has completed the RDMA Get. In a scenario where the receiver is busy doing computation and the sender wants to communicate, a *MPI_Send* posted by the sender will not block because all it has to do is to copy the data to be sent in an internal DMA buffer and send the information, needed by the receiver to get the data, through a SMSG packet. On the other hand, the receiver will check the CQ when a MPI function is invoked (like an *MPI_Wait*). After that, the receiver will be able to perform a RDMA Get (using FMA or BTE, according to the message size) and get the data from the sender.

Even though this scheme still allows overlapping, the actual transfer may require CPU involvement on the receiver side.

**R0 - RDMA Get** When the message size exceeds the Eager threshold (by default 8192 bytes), the R0 protocol is used. The mechanisms is pretty similar to E1 except that no DMA buffers are used. This means that R0 does not require extra data copies but it is much more sensitive to relative alignments of send/recv memory regions. Furthermore, because no internal buffers are used, an *MPI_Send* posted by the sender will block in order to prevent the data to be overwritten by the application. On the receiver side, because the transfer happens when an MPI function is called (MPI progress), the sender will wait for the entire duration of the computation performed by the receiver.

**R1 - RDMA Put** Messages bigger than MPI_GNI_MAX_NDREG_SIZE bytes (4 MByte by default) are sent through the R1 protocol based on a direct RDMA Put. Before the transmission, the sender sends a SMSG packet containing information needed to expose the memory to be filled. Then, the receiver sends a message back to the sender containing the data needed for the transfer. At this point, the sender performs an RDMA Put (using BTE) and sends a message to the receiver, informing that the transfer has been done. If the message is particularly big, this process can be repeated several times, until the whole message has been delivered. Like R0, this protocol does not allow computation/communication overlap.

Summarizing, the eager protocols usually require CPU involvement during the transfer (FMA) and thus they do not provide full asynchronous MPI message progress but can provide some sort of overlapping because of the small message sizes involved.

On the other hand, rendezvous protocols usually use the BTE method which allows one to perform data transfers without CPU involvement. Unfortunately, they still require the MPI library to be invoked in order to start the transfer.

As stated in Section 4, even though the network is capable of performing RDMA data transfer without using the CPU (BTE), the actual communication may not happen because of the MPI protocols involved (which require the user process to invoke the MPI library).

In order to overcome these issues, MPICH provides a *progress thread*. During the *MPI_Init* one thread is spawn for each MPI process with the aim of continuously check the CQ and get the SMSGs as soon as they arrive. The progress thread can poll the CQ continuously or wait for an interrupt. The performance differences between these two approaches are described by Hoefler et al. in [17].

### 6.3.2 Overlap in Passive MPI One-Sided Routines

Despite the capability exposed by the Aries interconnect, CrayMPI based on uGNI most likely implements one-sided MPI routines on top of a send/recv semantic. In other words, all the RMA operations are implemented in software; computation/communication overlapping can be achieved only using an helper thread running in background.

We think that such implementation, which does not make use of any RMA capability provided by the network, is because Cray MPI derives directly from MPICH. In fact, MPICH implements the one-sided routines on top of a queue-based mechanism which fits nicely the Completion Queue mechanism described in Section 6.1.1.

A better idea in terms of overlap would have been implementing PUT/GET/AMOs operations directly onto the RDMA capability of FMA DM and BTE, according to the message size. The developers' repository of the Open MPI project, currently provides such implementation and it is able to achieve computation/communication overlapping for the MPI pre-defined datatypes.

### 6.3.3 Enabling Progress Thread on Cray XC

In order to enable the MPI progress thread on Cray XC (needed by the default version of Cray MPI, based on uGNI, for getting asynchronous progress), some environment variables should be set as follows:

1. MPICH_NEMESIS_ASYNC_PROGRESS = [SC—MC]
2. MPICH_MAX_THREAD_SAFETY = multiple
3. MPICH_GNI_ASYNC_PROGRESS_TIMEOUT = 0

The first environment variable tells Cray MPI to use a progress thread. SC works on Gemini and Aries, whereas MC works only on Aries; the latter makes more efficient use of all the available virtual channels for asynchronous progress.

The second environment variable tells Cray MPI to use a thread safe MPI implementation. In fact, because more than one thread (at least two: master and progress) may invoke MPI at the same time, a thread safe version is required.

The third environment variable, if available, is not documented in the *mpi_intro* manual of cray-mpich/7.2.5. Anyway, the default value of -1 associated with this variable prevents the correct message progression.

### 6.4 Cray MPI based on DMAPP

The Distributed Memory Applications (DMAPP) APIs have been developed to support one-sided program models on Cray systems. The APIs exposed are intended to be used by libraries and compilers which want to realize the hardware performance of the underlying network technology. DMAPP does not exposed directly FMA and BTE mechanisms, but picks automatically the best method based on internal parameters.

Cray SHMEM, Cray UPC and Cray CAF are based on such APIs; Cray MPI provides a MPI implementation based on DMAPP, alternative to the default uGNI-based. The switch between the default and DMAPP version is based on the value of the environment variable `MPICH_RMA_OVER_DMAPP`. The DMAPP-based version of Cray MPI is able to execute contiguous PUTs and GETs in hardware without any interactions on the target side; accumulates and noncontiguous operations are executed in software with asynchronous progress based on progress thread (interrupt-based).

Summarizing, on a Cray XE/XC, the best way to get computation/communication overlapping with MPI is through the DMAPP-based version of Cray MPI, which uses the native RDMA hardware capability for implementing the MPI one-sided routines.

### 6.5 Alternative DMAPP-based MPI Implementations

In [12], Gerstenberger et al. present fast one-sided MPI (foMPI): a fully-functional MPI-3.0 RMA library implementation for Cray Gemini (XK5, XE6) and Aries (XC30) systems. The foMPI implementation assumes only small bounded buffer space at each process, no remote software agent, and only put, get, and some basic atomic operations for remote memory access, which is true for most RDMA hardware. In order to maximize asynchronous progression and minimize overhead, foMPI interfaces to the lowest available hardware APIs. For inter-node (network) communication, foMPI uses the DMAPP (Distributed Memory Application), which has direct access to the hardware (GHAL) layer. For intra-node communication, it uses XPMEM, a portable Linux kernel module that allows to map the memory of one process into the virtual address space of another.

In [4], Belli and Hoefler present foMPI-NA: a foMPI extension which includes and presents Notified Access. Notified Access is a new programming model in which the target of an RMA transfer may be notified (if needed) of remotely instantiated RMA transfers. According to the authors, such new approach provides the following benefits:

- It minimizes the number of messages needed by the synchronization phase, improving inter-node synchronization latencies and reducing the number of messages that cross the network.
- It enables zero-copy transfers saving energy and reducing cache pollution problems due to unnecessary data movements.
- It provides an increased asynchronous progression of processes and an augmented computation/communication overlap.
- It enables data-flow settings at the cost of a lightweight notification-matching mechanism.

The foMPI-NA implementation utilizes the uGNI API that provides direct access to the Cray's Fast Memory Access (FMA) and Block Transfer Engine (BTE) mechanisms. With both mechanisms is possible to directly notify the completion of an RDMA operation to the target process thanks to the use of uGNI completion queues.

# 7 Infiniband

The InfiniBand Architecture is an industry standard which defines communication and infrastructure for interprocessor communication and I/O. InfiniBand supports both the message and memory semantics explained in Section 3. In memory semantics, InfiniBand supports RDMA operations such as RDMA write and RDMA read. RDMA operations are one-sided and do not incur in software overhead at the other side. Like for the Cray Gemini/Aries interconnect, implementing the MPI one-sided routines using the RDMA capabilities exposed by the network is a good idea in order to obtain effective computation/communication overlapping.

Unfortunately, Infiniband is able to perform RDMA operations only on contiguous data. Noncontiguous data transfers and accumulate operations require the participation of the target process to perform an unpacking stage from a contiguous receiving buffer into the noncontiguous target location. In [26], Schneider et al. demonstrate a simple scheme for statically optimizing non-contiguous RMA transfers by combining partial packing, communication overlap, and remote access pipelining.

All the tests reported in this section have been run on Comet and Stampede: Comet is a Dell cluster hosted at San Diego Supercomputer Center equipped with Hybrid Fat-Tree, FDR InfiniBand. Stampede is a Dell PowerEdge C8220 Cluster hosted at Texas Advanced Computing Center equipped with a FDR InfiniBand.

## 7.1 Passive MPI One-Sided Routines Designs

In [19], Jiang et al. present remarkable results achieved using an RDMA-based implementation of active MPI one-sided routines.

The main challenge in extending the RDMA-based implementation from active mode to passive mode, is designing an efficient synchronization mechanism that allows one-sided operations to be performed at the target node independently of what the target process is doing. In other words, even though the RDMA transfer can be performed without any target involvement, the compliance with the MPI standard (lock/unlock) must be implemented and respected.

In [20], Jjiang et al. focus on the more challenging task of implementing RDMA-based passive MPI one-sided routines. They propose four alternative designs, two based on helper thread and two based on the atomic operations provided by Infiniband. They finally conclude that the atomic operation-based designs require less synchronization overhead, achieve better concurrency and consume fewer computing resources compared with the threads-based designs.

Anyway, the atomic operation-based have the drawback of high network traffic caused by repeated issue of atomic operations (remote polling).

In [25], Santhanaraman et al. propose a MCS like atomic-based design for exclusive locking in MPI (improving a design proposed in [20]). Unfortunately, they revert to a two-sided based approach for shared locking.

In [21], Li et al. propose a novel design for implementing truly passive shared and exclusive locking using Infiniband atomics. The idea is to implement a distributed queue using four data structures per window.

### 7.1.1 Overlap with State-of-the-art Libraries

The two most important MPI implementations taking advantage of the Infiniband technology are MVAPICH2 and Open MPI.

Despite all the designs presented in Section 7.1, both libraries currently implement passive synchronization primitives on top of two-sided operations.

In order to measure the impact of such implementation, we ran a simple overlap test on Comet and Stampede, both equipped with a FDX Infiniband interconnect able to offload the RDMA operations. The source code of the test is reported in Code 2 and Code 3, representing the origin (process 0) and target process (process np-1), respectively.

In case of optimal overlap, *comm_time* and *comp_time* are supposed to take approximately the same time; in fact, the sender should start the asynchronous transfer during the *MPI_Put* with minimal impact on the execution time, sleep for two seconds and then issue the *MPI_Win_Flush* which should be already completed. On the receiver side, the process sleeps for one second (during the *MPI_Put*) and then prints part of the array that is supposed to be overwritten by the sender process. Then it sleeps for two more seconds, simulating more computation on the receiver side.

For our analysis we used MVAPICH2-2.2b, compiled with the `--with-device=ch3:mrail --with-rdma=gen2` flags.

Code 2: Origin process

```
/* MPI_Win_lock_all common to origin and target */
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);
s_c = MPI_Wtime();
MPI_Put(v, n_elem, MPI_DOUBLE, np-1, 0, n_elem, MPI_DOUBLE, win);
usleep(2000000);
MPI_Win_flush(np-1, win);
e_c = MPI_Wtime();
comm_time = e_c - s_c;
```

Code 3: Target process

```
/* MPI_Win_lock_all common to origin and target */
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);
usleep(1000000);
printf(``Before flush on sender: %lf\n'',v[3]);
s_c = MPI_Wtime();
usleep(2000000);
e_c = MPI_Wtime();
comp_time = e_c - s_c;
```

During the experiment we noticed a nearly optimal overlap starting from the second data transfer. During the first transfer, no overlap was obtained. This shows pretty well the effect of a send/receive based passive synchronization. In fact, during the *MPI_Win_lock_all*, the shared lock request is only queued and it is performed during the first *MPI_Win_Flush*, implementing the lazy protocol explained in Section 5.2. The lock request is implemented on top of send/recv operations, which require the target process to call the MPI library. Once the (shared) lock is granted, all the other transfers can be implemented directly on top of the RDMA operations, without any participation of the target process.

For non-contiguous data types, as mentioned in Section 7, RDMA operations cannot be performed in hardware, thus no overlap can be obtained without a remote agent (helper thread).

## 8 Conclusions and Future Work

In this work we aimed to explain what computation/communication overlapping means and how it can be achieved on modern network interconnects, using the Message Passing Interface (MPI) library.

We also explained how the RDMA technology provided by several, widely used networks, is currently exploited by the state-of-the-art MPI libraries. In particular, we showed that these libraries, despite several designs proposed by various researchers, still rely on top of a send/receive approach for implementing passive one-sided communication routines. Furthermore, the RDMA capabilities exposed by Cray Aries and Infiniband can be used only on contiguous segments. When using user-defined data types in MPI, the target process needs to be involved in order to unpack and place the data in the right place. Only Cray MPI based on DMAPP is able to automatically manage this situation using an interrupt-based helper thread.

Despite the huge potential of overlapping communication with computation, MPI implementations still prefer to implement a weak interpretation of the Progress Rule, mainly because most applications do not expose enough opportunity for exploiting asynchronous transfers. Anyway, the one-sided functionalities described in the MPI-3 standard will pretty much force the issue of asynchronous progress on all MPI implementations [7].

## Acknowledgments

We gratefully acknowledge the support we received from the following institutions: National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, for the access on Hopper/Edison/Cori under the grant OpenCoarrays.
The Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575, for the access on Stampede and Comet.

## References

[1] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC series network. Cray Inc., White Paper WP-Aries01-1112, 2012.

[2] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 2010 IEEE 18th Annual Symposium on High Performance Interconnects*, HOTI '10, pages 83–87, Aug 2010.

[3] M. Beck and M. Kagan. Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proc. of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, DC-CaVES '11, pages 9–15. ACM, 2011.

[4] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*. IEEE, May 2015.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008.

[6] R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *Proc. of 18th Ann. Int'l Conf. on Supercomputing*, ICS '04, pages 298–305. ACM, 2004.

[7] M. Brinskiy and A. Supalov. Mastering performance challenges with the new mpi-3 standard. *Graduate from MIT to GCC Mainline*, page 33, 2014.

[8] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson. On the suitability of MPI as a PGAS runtime. In *Proc. of 2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[9] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 103:1–103:9, 2012.

[10] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. Opencoarrays: Open-source transport layers supporting coarray fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 4:1–4:11, 2014.

[11] A. Gavrilovska. *Attaining high performance communications: a vertical approach*. CRC Press, 2009.

[12] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, 2013.

[13] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, Nov. 2014.

[14] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[15] J. Hammond, S. Ghosh, and B. Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 44–58. Springer International Publishing, 2014.

[16] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. D. Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface*, volume 6305 of *LNCS*, pages 50–61. Springer, 2010.

[17] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *Proc. of 2008 IEEE Int'l Conf. on Cluster Computing*, pages 213–222, Sept. 2008.

[18] InfiniBand Trade Association and others. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[19] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '04, pages 531–538, April 2004.

[20] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. D. Gropp. Efficient implementation of MPI-2 passive one-sided communication on InfiniBand clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 68–76. Springer, 2004.

[21] M. Li, S. Potluri, K. Hamidouche, J. Jose, and D. K. Panda. Efficient and truly passive mpi-3 rma using infiniband atomics. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 91–96, 2013.

[22] MVAPICH2: High Performance MPI over InfiniBand and iWARP. Ohio State University.

[23] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network (QsNet): high-performance clustering technology. In *Hot Interconnects 9*, pages 125–130, 2001.

[24] H. Pritchard, I. Gorodetsky, and D. Buntinas. A uGNI-Based MPICH2 nemesis network module for the Cray XE. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 110–119. Springer Berlin Heidelberg, 2011.

[25] G. Santhanaraman, S. Narravula, and D. Panda. Designing passive synchronization for MPI-2 one-sided communication to maximize overlap. In *Proc. of IEEE 2008 International Symposium onParallel and Distributed Processing*, IPDPS '08, pages 1–11, April 2008.

[26] T. Schneider, R. Gerstenberger, and T. Hoefler. Compiler optimizations for non-contiguous remote data movement. In *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing*, Sep. 2013.

[27] C. Seitz. Myrinet–a gigabit-per-second local-area network. In *Hot Interconnects II, 1994. Symposium Record*, pages 161–180, 1994.

[28] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An asynchronous progress model for mpi rma on many-core architectures. In *Proc. of 29th Int'l Parallel and Distributed Processing Symposium*, IPDPS '15, 2015.

[29] M. ten Bruggencate and D. Roweth. DMAPP - an API for one-sided program models on Baker systems. In *Proceedings of Simulation Comes of Age*, CUG '10, 2010.

[30] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.

[31] X. Zhao, G. Santhanaraman, and W. Gropp. Adaptive strategy for one-sided communication in mpich2. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 16–26, Berlin, Heidelberg, 2012. Springer-Verlag.