Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures

Min Si,* Antonio J. Peña,† Jeff Hammond,‡ Pavan Balaji,† Masamichi Takagi,§ Yutaka Ishikawa§

*University of Tokyo, Japan msi@il.is.s.u-tokyo.ac.jp
†Argonne National Laboratory, USA {apenya, balaji}@mcs.anl.gov
‡Intel Labs, USA jeff_hammond@acm.org
§RIKEN AICS, Japan {masamichi.takagi, yutaka.ishikawa}@riken.jp

***Abstract*—In this paper we present "Casper," a process-based asynchronous progress solution for MPI one-sided communication on multi- and many-core architectures. Casper uses transparent MPI call redirection through PMPI and MPI-3 shared-memory windows to map memory from multiple user processes into the address space of one or more ghost processes, thus allowing for asynchronous progress where needed while allowing native hardware-based communication where available. Unlike traditional thread- and interrupt-based asynchronous progress models, Casper provides the capability to dedicate an arbitrary number of ghost processes for asynchronous progress, thus balancing application requirements with the capabilities of the underlying MPI implementation. We present a detailed design of the proposed architecture including several techniques for maintaining correctness per the MPI-3 standard as well as performance optimizations where possible. We also compare Casper with traditional thread- and interrupt-based asynchronous progress models and demonstrate its performance improvements with a variety of microbenchmarks and a production chemistry application.**

***Keywords*-MPI ghost process; one-sided communications; RMA; multi-core; many-core; asynchronous progress;**

I. Introduction

The MPI-2 and MPI-3 standards [1] introduced one-sided communication semantics (also known as remote memory access or RMA) that allow one process to specify all communication parameters for both the sending and receiving sides. Thus, a process can access memory regions of other processes in the system without the target process explicitly needing to receive or process the message. RMA provides an alternative model to traditional two-sided or group communication models and can be more natural for some applications to use [2].

While the RMA model is useful for a number of communication patterns, the MPI standard does not guarantee that such communication is asynchronous. That is, an MPI implementation might require the remote target to make MPI calls in order to ensure communication progress to complete any RMA operations issued on it as a target. This requirement is common in many MPI implementations. Specifically, most network interfaces do not natively support complex one-sided communication operations. For example, networks such as InfiniBand provide contiguous PUT/GET operations only in hardware. Thus, any contiguous PUT/GET MPI RMA communication can be implemented almost entirely in hardware, allowing the hardware to fully handle its progress semantics. On the other hand, if the application developer wants to do an accumulate operation on a 3D subarray, for instance, such an operation must be done in software within the MPI implementation. Consequently, the operation cannot complete at the target without explicit processing in software and thus may cause arbitrarily long delays if the target process is busy computing outside the MPI stack.

To ensure asynchronous completion of operations (i.e., "asynchronous progress"), traditional implementations have relied on two models. The first is to utilize background threads [3] dedicated to each MPI process in order to handle incoming messages from other processes (e.g., used by most MPI implementations including MPICH [4], MVAPICH [5], Intel MPI [6], and IBM MPI on Blue Gene/Q [7]). While this model is a generic approach for all communication models, it has several drawbacks. For example, it is limited with the address space sharing that the operating system provides. Specifically, a background thread can make progress for only one MPI process (i.e., the process to which it belongs) and thus requires at least as many background threads as the number of MPI processes. On current MPI implementations, where the MPI library polls on the network to look for messages, this approach can waste half the hardware threads or cores. Furthermore, this model forces multithreaded communication overhead on all MPI operations, which can be expensive [8].

The second model is to utilize hardware interrupts to awaken a kernel thread and process the incoming RMA messages within the interrupt context (e.g., used by Cray MPI [9] and IBM MPI on Blue Gene/P [10, Chapter 7]). While this model does not have the large number of wasted cores or multithreaded communication overhead of the background thread approach, it is limited in that each message that requires target-side processing generates an interrupt, which can be expensive. Perhaps more fundamental, the model of hardware interrupts is centered on the notion that all hardware resources are busy with user computation, which needs to be *interrupted* to inject computation related to asynchronous progress. We believe this is not the right

point of view for modern many-core architectures where the number of cores is very large.

In this paper we present "Casper," a process-based asynchronous progress solution for MPI on multi- and many-core architectures. An alternative to traditional thread- and interrupt-based models, Casper provides a distinct set of benefits for applications, which we believe are more appropriate for large many-core architectures. Unlike traditional approaches, the philosophy of the Casper architecture is centered on the notion that since the number of cores in the system is growing rapidly, dedicating some of the cores for helping with asynchronous progress might be better than using an interrupt-based shared-mode model. Similarly, the use of processes rather than threads allows Casper to control the amount of sharing, thus reducing thread-safety overheads associated with multithreaded models, as well as to control the number of cores being utilized for asynchronous progress.

The central idea of Casper is the ability of processes to share memory by mapping a common memory object into their address spaces by using the MPI-3 shared memory windows interface. Specifically, Casper keeps aside a small user-specified number of cores on a multi- or many-core environment as "ghost processes." When the application process tries to allocate a remotely accessible memory window, Casper intercepts the call and maps such memory into the ghost processes' address space. Casper then intercepts all RMA operations to the user processes on this window and redirects them to the ghost processes instead.

Since the user memory regions are not migrated or copied but just mapped into the ghost processes' address space, RMA operations that are implemented in hardware see no difference in the way they behave. On the other hand, RMA operations that require remote software intervention can be executed in the ghost processes' MPI stack on the additional cores kept aside by Casper, without requiring any intervention from the application processes.

Although the core concept of Casper is straightforward, the design and implementation of such a framework must take several aspects into consideration. Most important, the framework needs to ensure that correctness is maintained as required by the MPI-3 semantics. While this task is easy to manage for simple applications, the wide variety of communication and synchronization models provided by MPI can make the task substantially more complex for applications that are nontrivial. This task is even more complicated for applications that use multiple MPI-3 epoch types (e.g., passive-target and active-target) or multiple windows of remote memory buffers because the same Casper ghost processes need to maintain progress on all of them, thus essentially requiring that they never indefinitely block inside an MPI operation. Furthermore, when more than one ghost process is present, the Casper architecture must ensure that the ordering, atomicity, and memory consistency requirements specified by the MPI-3 standard are met in a way that is transparent to the application.

The Casper architecture hides all this complexity from the user and manages it internally within its runtime system. In some cases, however, such complexity can cause performance overhead. In this paper we present various techniques we used to ensure correctness while retaining the performance of the RMA operations and enabling low-overhead asynchronous progress. In addition to a detailed design of the Casper architecture, we present experiments evaluating and analyzing Casper with various microbenchmarks and a large quantum chemistry application.

Recommended reading: While this paper provides some information on the MPI RMA semantics, it is not meant to be a comprehensive description. In order to better understand the subtle characteristics and capabilities of MPI RMA on which this paper relies, we highly recommend reading past papers and books that more thoroughly discuss these semantics (e.g., [11], [12]).

II. Casper Design Overview

Casper is designed as an external library through the PMPI name-shifted profiling interface of MPI. This allows Casper to transparently link with various MPI implementations, by overloading the necessary MPI functions. Casper provides three primary functionalities: (1) deployment of ghost processes to help with asynchronous progress, (2) RMA memory allocation and setup, and (3) redirection of RMA communication operations to appropriate ghost processes.

A. Deployment of Ghost Processes

Ghost processes in Casper are allocated in two steps. In the first step, when the user launches the application with a number of processes, a user-defined subset of these processes is carved aside as the ghost processes at MPI initialization time (see Figure 1). The remaining processes form their own subcommunicator called **COMM_USER_WORLD**. The number of ghost processes is user-defined through an environment variable, allowing the user to dedicate an arbitrary number of cores on the node for the ghost processes.

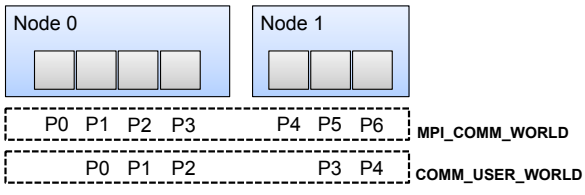


Figure 1. Casper ghost process management.

In the second step, Casper overrides all MPI operations that take a communicator argument and replaces any occurrence of **MPI_COMM_WORLD** in all non-RMA functions with **COMM_USER_WORLD** at runtime through PMPI redirection. This step ensures that all non-RMA communication

is redirected to the correct MPI processes, including creation of other subcommunicators from **MPI_COMM_WORLD**.

After initialization, ghost processes simply wait to receive any commands from user processes in an **MPI_RECV** loop. This approach ensures that while the ghost processes are waiting for commands, they are always inside the MPI runtime, thus allowing the MPI implementation to make progress on any RMA operations that are targeted to those ghost process.

One aspect to consider in Casper is the locality of application buffers relative to the ghost processes. Specifically, since a ghost process might be depositing or reading data from the application buffers, how far the ghost process is compared with the buffers can have a serious impact on performance. To handle this issue, we ensure that the ghost processes in Casper are topology-aware. Casper internally detects the location of the user processes and places its ghost processes as close to the application process memory as possible. For example, if a node has two NUMA domains and the user requests two ghost processes, each of the ghost processes places itself in a different NUMA domain and binds itself to either the process ranks or segments in that NUMA domain.

B. RMA Memory Allocation and Setup

Remote memory allocation in the Casper architecture is tricky in that the allocated memory must be accessible by both the application processes and the ghost processes. MPI provides two broad mechanisms to declare a memory region as remotely accessible. The first is an "allocate" model (i.e., **MPI_WIN_ALLOCATE** and **MPI_WIN_ALLOCATE_SHARED**) in which MPI is responsible for creating such memory, thus allowing the MPI implementation to optimize such allocation (e.g., through shared memory or globally symmetric virtual memory allocation). The second is a "create" model (i.e., **MPI_WIN_CREATE** and **MPI_WIN_CREATE_DYNAMIC**), in which the user allocates memory (e.g., using `malloc`) and then exposes the memory as remotely accessible.

While memory sharing between the application processes and the ghost processes can occur in both models, doing so in the "create" model requires OS support to expose such capability. This capability is generally present on large supercomputers such as Cray (e.g., through XPMEM [13] or SMARTMAP [14]) and Blue Gene, but not always on traditional cluster platforms. Thus, for simplicity, we currently support only the "allocate" model.

When the application creates an RMA window using **MPI_WIN_ALLOCATE**, Casper follows a three-step process:

1) It first allocates a shared-memory region between the user processes and the ghost process on the same node using the MPI-3 **MPI_WIN_ALLOCATE_SHARED** function, as depicted in Figure 2. As shown in the figure, the same memory region that is used by the application is also mapped onto the address space of the ghost process. Thus, such memory is accessible through either process, although care must be taken to keep it consistent.
2) Once the shared memory is allocated, it creates a number of internal windows using **MPI_WIN_CREATE** to expose this memory to all user and ghost processes.
3) Casper creates a new window with the same memory region that contains only the user processes; it then returns the new window handle to the application.

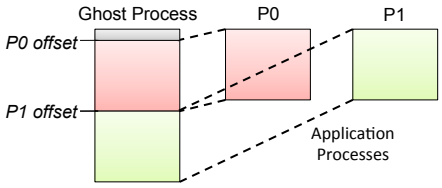


Figure 2. Casper RMA Buffer Mapping

We note that the Casper architecture exposes the allocated shared-memory in multiple overlapping windows. This model provides Casper's runtime system with enough flexibility to manage permissions and communication aspects in a highly sophisticated manner; but at the same time the model requires extreme caution to ensure that memory is not corrupted and is consistent with the user's expectation. In Section III, we describe how these internal windows are utilized in Casper.

C. RMA Operation Redirection

Once the window becomes ready, Casper transparently redirects, through PMPI redirection, all user RMA operations to the ghost processes on the target node. Such redirection needs to translate both the target rank that the RMA operation is addressed to and the target offset where the data needs to be written to or read from (since the offset in the ghost process's memory region might not be the same as the offset in the user process's memory region). For example, based on Figure 2, if an origin process does an RMA operation at offset "X" of user process P1, Casper will redirect the operation to offset "X + P1's offset in the ghost process address space" on the ghost process.

When multiple ghost processes are available on the target node, Casper attempts to utilize all of them by spreading communication operations across them. This approach allows the software processing required for these operations to be divided between the different ghost processes, thus improving performance. Using such a model with multiple ghost processes, however, requires extra care compared with using a model with a single ghost process. Moreover, it raises a number of correctness issues, as we discuss in Section III-B.

III. Ensuring Correctness and Performance

In this section we discuss several cases that we need to handle inside Casper in order to maintain correctness as specified in the MPI-3 standard while achieving high performance.

With respect to performance optimizations, some of the proposed optimizations are automatically detected and handled by the Casper implementation, while some others are based on user hints in the form of either *info* hints or *assert* hints, as specified by the MPI standard. Both *info* and *assert* hints are, in essence, user commitments to comply with different restrictions that allow the MPI implementation to potentially leverage different optimizations. Specifically, *info* hints are broad-sweeping and apply to an entire window and all operations issued on that window. Further, *info* hints are extensible so each MPI implementation can add newer hint capabilities to improve its own performance. *assert* hints, on the other hand, are more focused in scope and typically apply to each epoch. They are also not as easily extensible to MPI implementation-specific hints.

The *info* hints used in Casper are not defined by the MPI-3 standard and are Casper-specific extensions. In contrast, the *assert* hints used in Casper all are MPI-3 standard defined hints that we reuse with the same semantics as the standard. Thus, *hints* are compatible with other MPI implementations as well, even though some MPI implementations might not choose to take advantage of them.

A. Lock Permission Management for Shared Ghost Processes

Consider an environment where multiple application processes reside on different cores of the same node and thus share a ghost process. In this case, all RMA communication to these application processes would be funneled through the same ghost process. In such an environment if an origin wanted to issue an exclusive lock to more than one application process on the same node, such a step would result in multiple exclusive lock requests being sent from the origin to the same ghost process. This is disallowed by the MPI standard—an origin cannot nest locks to the same target. Similarly, if two origin processes issue exclusive locks to different application processes on the same node, this would result in multiple exclusive lock requests being sent from different origins to the same ghost process. While this is correct according to the MPI standard, it would result in unnecessary serialization of all exclusive locks to processes on the same node, thus hurting performance significantly.

To overcome this issue, Casper internally maintains separate overlapping windows for each user process on the node. In other words, if a ghost process is supporting N user processes, it will create N overlapping windows. Communication to the ith user process on each node goes through the ith window. Thus, the number of internal overlapping windows created is equal to the maximum number of user processes on any node of the system. Such overlapping windows allow Casper to carefully bypass the lock permission management in MPI when accessing different processes but to still take advantage of them while accessing the same process. Since a single RMA communication operation cannot target multiple processes at the same time, we never run into a case where the bypassing of permission management across processes causes an issue.

While this approach ensures correctness, it can be expensive for both resource usage and performance. To alleviate this concern, we allow the user to use the info hint `epochs_used` to specify a comma-separated list of epoch types that the user intends to use on that window. The default value for this info key is all epoch types (i.e., "fence,pscw,lock,lockall"); but if the user sets this value to a subset that does not include "lock," Casper can use that information to create only a single overlapping window (apart from the user-visible window) for all its internal operations and reduce any overhead associated with lock permission management.

B. Managing Multiple Ghost Processes

In Casper, the user is allowed to configure a node with multiple ghost processes. Doing so allows better sharing of work when the number of operations requiring such asynchronous progress is large. However, such a configuration requires additional processing to maintain correctness. A simple model in which all communication is randomly distributed across the different ghost processes has two issues that need to be handled: (1) lock permissions in the *lock-unlock* epoch and (2) ordering and atomicity constraints for accumulate operations.

When the *lock-unlock* epoch type is used, Casper will internally lock all ghost processes on a node, when a lock operation for a particular application process is issued, in the hope of spreading communication across these ghost processes. In practice, however, many MPI implementations might not acquire the lock immediately, instead delaying them to a future time (e.g., when an RMA communication operation is issued to that target). Given this behavior, consider an application that simply does one *lock-put-unlock*. In this example, Casper might randomly pick a ghost process, thus picking one ghost process on one origin while picking a different ghost process on another origin. For implementations that delay lock acquisition, this example would mean that the two ghost processes would get exclusive locks from two different origins to access the same memory location. Since the lock management in MPI is unaware of the shared-memory buffers in Casper, both exclusive locks would be granted, resulting in data corruption.

The second issue concerns the atomicity and ordering guarantees provided by the MPI standard for concurrent accumulate operations to the same location (see [1], Section

11.7.1). Each basic datatype element of concurrent accumulate operations issued by the same or different origin processes to the same location of a target process must be performed atomically. Similarly, two accumulate operations from the same origin to the same target at the same memory location are strictly ordered. In Casper, if a user process is served by a single ghost process, such atomicity is already provided by the MPI implementation. If a user process is served by multiple ghost processes, however, they might simultaneously be accessing the same memory region, thus breaking both atomicity and ordering.

To address these issue, Casper uses a two-phase solution. The first phase is to provide a base "static binding" model in which each ghost process is statically assigned to manage only a subset of the remotely accessible memory on the node. This model ensures correctness as per the MPI standard but can have some performance cost. We propose two static binding approaches in this paper: rank binding and segment binding. The second phase is to identify periods in the application execution where the issuing of some operations to ghost processes can be done in a more dynamic fashion. In this section, we discuss both phases.

1) Static Rank Binding: With rank binding, each user process binds to a single ghost process, and any RMA operations issued to that user process are always directed to that ghost process. Therefore, different origins locking the same target are redirected to the same ghost process, thus benefiting from MPI's internal permission management. Similarly, different accumulate operations targeting the same user process are redirected to the same ghost process, thus benefiting from MPI's internal ordering and atomicity management. This model completely works around the problem with multiple ghost processes since each user application process is associated with only a single ghost process. The disadvantage of this approach, however, is that if the amount of communication to the different user application processes is not uniform, one ghost process might get more work than the others get, thus causing load imbalance.

2) Static Segment Binding: With segment binding, the total memory exposed by all the processes on the node is segmented into as many chunks as the number of ghost processes, and each chunk or segment is bound to a single ghost process. Thus, given a particular byte of memory, a single ghost process "owns" it. When the user application issues a lock operation, Casper will still lock all ghost processes; but when the actual RMA communication operation is issued, it is redirected to the appropriate ghost processes that own that segment of the memory. In this model different origin processes can get simultaneous access to the same target through different ghost processes. However, they cannot simultaneously update the same memory region, thus making such shared access inconsequential and still guaranteeing application correctness.

A second aspect that must be considered in segment binding is that segmentation must be at a basic-datatype-level granularity in order to maintain MPI's requirements for atomicity. To handle this, we must ensure that segments are divided at an alignment of the maximum size of MPI basic datatypes (i.e., 16 bytes for `MPI_REAL`). This alignment is needed in order to guarantee that no basic datatype is divided between two ghost processes. Thus, although an operation may be divided into multiple chunks and issued to different ghost processes, each basic datatype unit belongs to a single chunk and is directed to a single ghost process, thus guaranteeing atomicity and ordering. This approach will work in most cases since most compilers enable data alignment by default (i.e., a double variable has to be allocated on an address that is a multiple of eight). Hence, it is safe to divide an operation into different aligned segments. We note, however, that this approach is not strictly portable. Compilers are allowed to not enforce data alignment or allow users to explicitly disable structure padding, resulting in unsafe segmentation. Nevertheless, data alignment is always recommended for performance; and some architectures, such as SPARC [15], even require it for correctness.

The advantage of the static segment binding model compared with the static rank binding model is that the load on a given ghost process is determined by the memory bytes it has access to, rather than the process it is bound to. In some cases, such a model can provide better load balancing than the static rank binding model. However, the static segment binding model has several disadvantages. Most important, this solution relies on analyzing the specific bytes on the target process that are being accessed for each RMA operation. For operations using contiguous datatypes that completely fall within one data segment, this model can be straightforward, since the operation is simply forwarded to the appropriate ghost process. If the data overlaps two or more segments, however, Casper must internally divide the operation into multiple operations issued to different ghost processes. This solution becomes even more complex when the data being transmitted is noncontiguous, in which case the datatype needs to be expanded and parsed before the segments it touches can be determined.

3) Dynamic Binding: In applications that have balanced communication patterns, each target process on a compute node tends to receive an approximately equal number of RMA operations. The best performance can be achieved for such patterns by equally distributing the number of processes handled by each ghost process. In such cases, a static binding approach might be a good enough solution for load balancing. For applications with more dynamic communication patterns, however, a more dynamic selection of ghost processes is needed, as long as such an approach does not violate the correctness requirements described above.

In Casper, to help with dynamic binding, we define "static-binding-free" intervals of time. For example, suppose

the user application issues a lock operation to a target. This lock would be translated to a lock operation to the corresponding ghost process to which the target process is bound. After issuing some RMA communication operations, if the user application flushes the target, the MPI implementation must wait for the lock to be acquired and cannot delay the process of lock acquisition any further. The period after the flush operation has completed and before the lock is released is considered a "static-binding-free" period. That is, in this period we know that the lock has already been acquired. In such periods, the Casper implementation no longer has to do lock permission management and is free to load balance PUT/GET operations to any of the ghost processes with the same lock type as that specified by the user application process. We note that this optimization is not valid for accumulate-style operations, in order to maintain the atomicity and ordering guarantees specified by the MPI standard.

We utilize three dynamic load-balancing approaches in Casper. The first is a "random" algorithm that randomly chooses a ghost process from the available ghost processes for each RMA operation. The second is an "operation-counting" algorithm that chooses the ghost process that the origin issued the least number of operations to. The third is a "byte-counting" algorithm that chooses the ghost process that the origin issued the least number of bytes to.

C. Dealing with Multiple Simultaneous Epochs

The MPI standard does not allow a process to simultaneously participate in multiple overlapping epoch types on a given window. However, for disjoint sets of processes or for the same set of processes with different windows, no such restrictions exist. Thus, one could imagine an application in which a few of the processes are participating in a *lock-unlock* epoch on one window, while another disjoint set of processes is participating in a *fence* epoch on another window. If more than one of these processes are on the same node, the ghost processes have to manage multiple simultaneous epochs. The primary difficulty with handling multiple simultaneous epochs, especially active target epochs such as *fence* and *PSCW*, is that the epoch opening and closing calls in these epochs are collective over either all or a subset of processes in the window and these calls are blocking with no nonblocking variants. Thus, if a ghost process participates in one epoch opening or closing call, it is stuck in a blocking call and hence loses its ability to help with other epochs for other user processes.

To work around this issue, Casper converts all active-target epochs into passive-target epochs on a separate window. Further, it manages permission conflicts between *lock-all* and *lock* by converting *lockall* to a collection of *lock* operations in some cases. The following paragraphs describe these changes in more detail.

1) Fence: The *fence* call supports a simple synchronization pattern that allows a process to access data at all processes in the window. Specifically, a fence call completes an epoch if it was preceded by another fence and starts an epoch if it is followed by another fence.

In Casper, we translate *fence* to a *lockall-unlockall* epoch. Specifically, we use a separate window for *fence*; and when the window is allocated, we immediately issue a *lockall* operation. When the user application calls *fence*, we internally translate it to *flushall-barrier*, where the *flushall* call ensures the remote completion of all operations issued by that origin and the *barrier* call synchronizes processes, thus ensuring the remote completion of all operations by all origins. This model ensures that the ghost processes do not need to explicitly participate in any active target synchronization calls, thus avoiding the blocking call issues discussed above.

While correct, this model has a few performance issues. First, a *fence* call does not guarantee remote completion of operations. The return of the *fence* call at a process guarantees only the local completion of operations issued by that process (as an origin) and the remote completion of operations issued to that process (as a target). This is a weaker guarantee than what Casper provides, which is remote completion of all operations issued by all processes. Casper's stricter guarantees, while correct, do cost performance, however. Therefore, such remote completion through *flushall* can be skipped if the user provides the `MPI_MODE_NOPRECEDE` assert indicating that no operations were issued before the *fence* call that need to be flushed.

Second, an MPI implementation can choose to implement *fence* in multiple different ways. For example, one possible implementation of the *fence* epoch is to delay all RMA communication operations to the end of the epoch and issue them only at that time. Thus, if the MPI implementation knows that a *fence* call does not complete any RMA communication operations (e.g., if it is the first fence), it can take advantage of this information to avoid synchronizing the processes. Casper does not have this MPI implementation internal knowledge, however. Thus, it always has to assume that the MPI implementation might issue the RMA communication operations immediately, and consequently it always has to synchronize processes. Again, doing so costs performance. However, if the user specifies the `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, and `MPI_MODE_NOPRECEDE` asserts, Casper can skip such synchronization since there are no store operations before the *fence* and no PUT operations after the *fence* that might impact the correctness of the data.

Third, when *fence* is managed by the MPI implementation, it internally enforces memory consistency through appropriate memory barriers. In Casper, since the *fence* call is translated to passive-target synchronization calls,

such memory consistency has to be explicitly managed. Thus, during each *fence* call, we add an additional call to **MPI_WIN_SYNC** to allow such memory ordering consistency, costing more performance.

2) PSCW: The *PSCW* epoch allows small groups of processes to communicate with RMA operations. It explicitly decouples calls in order to expose memory for other processes to access (exposure epoch) and calls to access memory from other processes (access epoch). The **MPI_WIN_POST** and **MPI_WIN_WAIT** calls start and end an exposure epoch, while the **MPI_WIN_START** and **MPI_WIN_COMPLETE** start and end an access epoch.

As with *fence*, we translate the *PSCW* epoch to passive-target synchronization calls on the same window (since *fence* and *PSCW* cannot simultaneously occur on the same window). Also as with *fence*, we add additional process synchronization for *PSCW* in Casper. Instead of using *barrier*, however, we use *send-recv* because the processes involved might not be the entire group of processes on the window. Consequently, *PSCW* encounters the same set of drawbacks as *fence* with respect to performance. To help with performance, we allow the user to provide the `MPI_MODE_NOCHECK` assert specifying that the necessary synchronization is being performed before *post* and *start* calls. When this assert is provided, Casper can drop additional synchronization.

3) Lockall: The *lockall* epoch is a passive-target epoch and thus does not require participation from the ghost processes. However, we need to be careful that we do not bypass lock permission requirements when the user uses both *lockall* and *lock* simultaneously from different origin processes. In this case, as discussed in Section III-A, since the *lock* calls are redirected to internal overlapping windows by Casper, one process of the application might end up acquiring a *lockall* epoch while another process of the same application acquires an exclusive-mode *lock* epoch on the same window (we note that the *lockall* epoch is shared-mode only and does not have an exclusive-mode equivalent). This situation is obviously incorrect and can cause data corruption.

To avoid this, Casper internally converts the *lockall* epoch to a series of locks to all ghost processes. Doing so ensures that any accesses are correctly protected by the MPI implementation. Arguably, this solution can add some performance overhead since it serializes lock acquisition. However, most MPI implementations delay lock acquisition until an actual operation is issued to that target, so this might not be much of a concern in practice.

D. Other Considerations

To maintain correctness, we also had to address several other aspects including self-locks (which are guaranteed by MPI to not be delayed, in order to support load/store operations) and memory load/store ordering consistency in the presence of multiple application and ghost processes. Because of space limitations, however, we do not describe them here.

IV. Evaluation

In this section, we evaluate Casper on two platforms: the NERSC Edison Cray XC30 supercomputer (https://www.nersc.gov/users/computational-systems/edison/configuration/) and the Argonne Fusion cluster (http://www.lcrc.anl.gov/about/fusion). We used these two platforms to demonstrate the impact of varying levels of hardware support for RMA operations. Specifically, Cray MPI (version 6.3.1) can be executed in two modes: regular or DMAPP-based. The regular version executes all RMA operations in software with asynchronous progress possible through a background thread. The DMAPP version executes contiguous PUTs and GETs in hardware, but accumulates and noncontiguous operations are executed in software with asynchronous progress through interrupts. On the Fusion platform, we used MVAPICH. MVAPICH (version 2.0rc1[1]) implements contiguous PUT/GET operations in hardware, while using software active messages for accumulates and noncontiguous operations (asynchronous progress using a background thread).

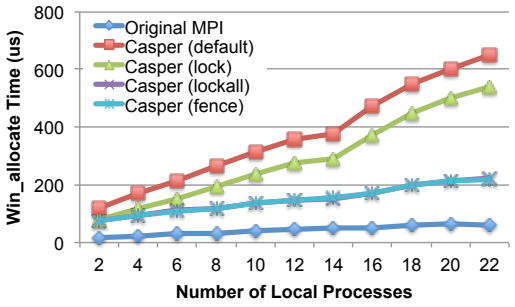
We expect Casper to improve asynchronous progress in the cases where RMA operations are implemented as *software active messages* and to perform as well as the original MPI implementation when hardware direct RMA is used.

A. Overhead Analysis

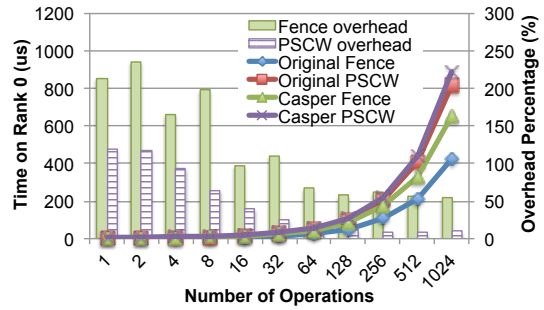
In this section, we measure two overheads caused by Casper: (1) window allocation and (2) Fence and PSCW.

As discussed in Section II-B, Casper internally creates additional overlapping windows in order to manage lock permissions when a ghost process supports multiple user processes. These can cause performance overhead. However, the amount of overhead can be controlled by setting the info argument `epoch_type` to tell Casper which epoch types are used by the application. Accordingly Casper can decide which internal windows it needs to create. Figure 3(a) shows the overhead of **MPI_WIN_ALLOCATE** on a user process with varying total numbers of processes on a single node of Cray XC30. When no info hints are passed (default `epoch_type` is "fence,pscw,lockall,lock"), Casper can experience substantial performance cost in window creation time. When `epoch_type` is set to "lock," Casper does not have to create the additional window for active target and lockall communication, thus improving performance a little; but the cost is still considerable because Casper has to create one window for every user process on that node. When `epoch_type` is set to "lockall" or "fence" (or any other

[1]We had to fix a bug in MVAPICH to allow for true hardware-based RMA for PUT and GET.



(a) Window allocation overhead.



(b) Fence and PSCW overhead

Figure 3. Overhead analysis.

value that does not include "lock"), Casper has to create just one additional internal window, thus reducing the cost substantially, although the cost is still more than twice that of original MPI.

The second major overhead occurs because of the conversion of fence and PSCW to passive-target epochs and the additional synchronization and memory consistency associated with it. We measure these overheads by using two interconnected processes on Cray XC30. The fence experiment performs *fence–accumulate–fence* on the first process and *fence–fence* on the other, with the first passing the `MPI_MODE_NOPRECEDE` assert and the second fence passing the `MPI_MODE_NOSUCCEED` assert. The PSCW experiment performs *start–accumulate–complete* and *post–wait* on the two processes. Figure 3(b) shows the execution time of our experiments on the first process. While the overhead is large (100–200%) for a small number of operations, as the number of operations issued increases, this cost gets amortized and disappears.

B. Asynchronous Progress

In the section, we demonstrate the asynchronous progress improvements achieved in various scenarios.

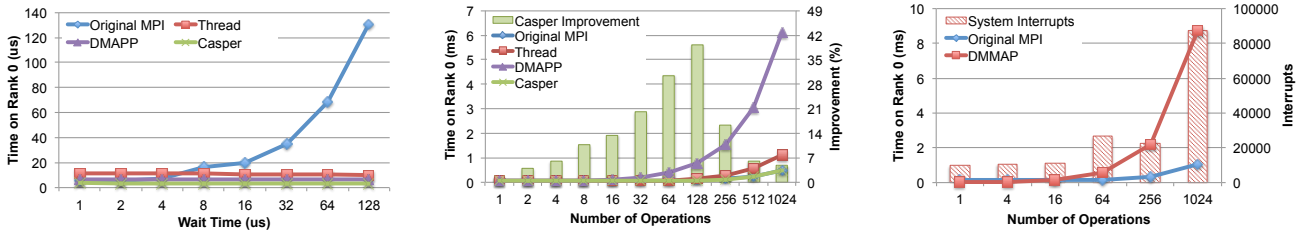
1) Different Synchronization Modes: Our first experiment demonstrates the improvement of computation and communication overlap in passive and active-target modes using Casper. Two interconnected processes are used in each mode. In the passive-target mode, one process issues *lockall–accumulate–unlockall* to another process while that process is blocking in computation. Figure 4(a) shows the results on the Cray XC30. As expected, with the original MPI the execution time on the origin increases with wait time on the target, which means that the origin is blocked by the computation on the target. All asynchronous progress approaches relieve this issue. We note, however, that both the DMAPP and thread approaches have more overhead than Casper does.

The overhead with using the MPICH asynchronous thread comes from the expensive thread-multiple safety and lock contention. DMAPP-based asynchronous progress, however, does not involve thread-multiple safety and also wakes up background threads only when a message arrives. Therefore, to analyze the reason for this overhead, we performed a test in which one process does *lockall–accumulate–unlockall* and the other process does a *dgemm* computation. As shown in Figure 4(c), when we increase the number of ACCUMULATEs issued in each iteration, DMAPP's overhead also increases. To further analyze this situation, we measured the number of system interrupts. We found that they increased with the number of ACCUMULATEs as well and were clearly becoming a bottleneck with increasing numbers of operations.

In the active-target mode, since both fence and PSCW require internal synchronization in Casper, the origin has to wait for the completion of the epoch on the target. Thus, in an experiment similar to that for the passive mode, we measured the time for *fence–accumulate–fence* on one process while another process performs *fence–100 μs busy waiting–fence* as shown in Figure4(b). We notice that when a small number of operations are issued during fence, asynchronous progress is beneficial. But when the communication takes more time than the delay on the target, which is the maximum time Casper can overlap (larger than 128 in the figure), the percentage improvement decreases, as expected. PSCW follows a similar trend. Both DMAPP and thread asynchronous progress show significant overhead compared with that of the original MPI execution.

2) Different RMA implementations: The second experiment focuses on the scalability of asynchronous progress with different RMA implementations. In this experiment every process communicates with all the other processes in a *communication–computation–communication* pattern. We use one RMA operation (size of a double) in the first communication, 100 μs of computation, and ten RMA operations (each size double) in the second communication.

On Cray XC30, we use one process per node and scale the number of nodes for both ACCUMULATE and PUT, as shown in Figures 5(a) and 5(b). We note that DMAPP enables direct RMA for PUT/GET with basic datatypes in Cray MPI, but it involves interrupts for ACCUMULATE operations. Consequently, Casper outperforms the other approaches for ACCUMULATE, while achieving the same



(a) Passive-target RMA (b) Fence RMA (c) Overhead of DMAPP interrupts

Figure 4. Overlap improvement using two interconnected processes on Cray XC30.

performance as that of DMAPP for PUT/GET. The thread asynchronous progress is always expensive and even worse than that of the original MPI when a large number of processes are communicating.

On the Fusion cluster, we compared Casper with MVAPICH by also using one process per node. Figure 5(c) indicates that Casper improves asynchronous progress for ACCUMULATE, which is still implemented with software active messages in MVAPICH. The thread asynchronous progress again shows significant overhead. We also measured the performance of PUT/GET operations; as expected, the performance of Casper was identical to that of original MPI since these operations are implemented directly in hardware. The performance numbers are not shown here because of space limitations.

C. Performance Optimization

Our third set of microbenchmarks focuses on the different load-balancing optimizations discussed in Section III-B.

1) Static Rank Binding: Figure 6 shows our measurements with static rank binding on Cray X30. In the first experiment (Figure 6(a)) we show the static rank binding with increasing number of processes when each process sends one accumulate message (size of double) to every other process in the system. We use 16 processes per node and evaluate Casper with up to 8 ghost processes on each node. Our results indicate that two ghost processes are sufficient when up to 32 processes communicate; when more processes communicate, however, configurations with larger numbers of ghost processes tend to perform better. The reason is that the number of incoming RMA operations increases with more processes, thus requiring more ghost processes computing to keep up.

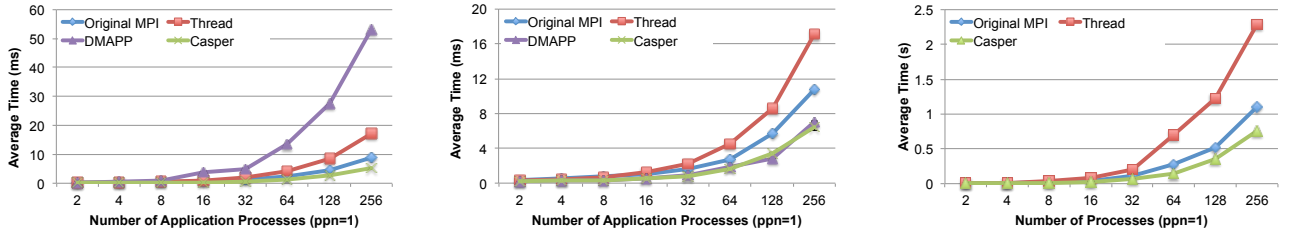
Figure 6(b) shows a similar experiment but increases the number of accumulate operations while keeping the user process count constant at 32 (2 nodes with 16 processes each). The results show a trend similar to that of the previous experiment, with more ghost processes benefiting when the number of operations per process is larger than 8.

2) Static Segment Binding: In this experiment we evaluate the performance of the static segment binding approach. Such an approach is expected to be especially beneficial when the application allocates uneven-sized windows and receives a large number of operations that need to be processed in software. Figure 6(c) demonstrates this pattern. We used 16 nodes with 16 processes and up to 8 ghost processes per node. The first process of every node allocates a 4-kilobyte window (512 count of double), while the others only allocate 16 bytes. Then each process performs a *lockall–accumulate–unlockall* pattern on all the other processes. We increase the number of ACCUMULATEs to each process whose local rank is 0 while issuing a single operation to other processes. As shown in the figure, performance improves with increasing numbers of ghosts, because the large window is divided into more segments and the communication issued to different segments is handled by different ghosts.

3) Dynamic Binding: To test our dynamic binding approaches, we designed three microbenchmarks, all of which are executed on 16 nodes with 20 user processes and 4 ghost processes per node.

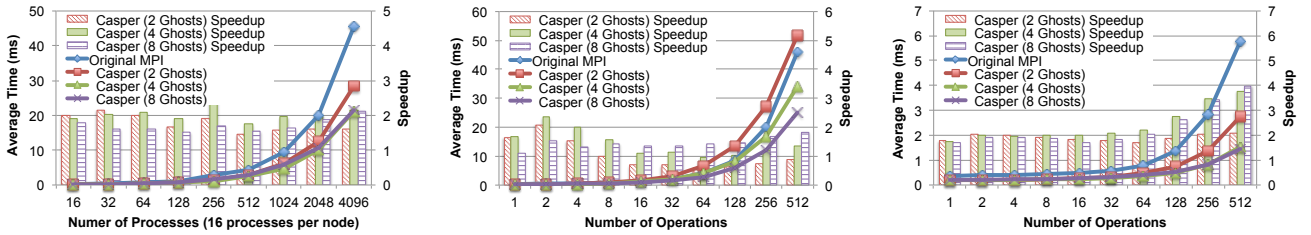
Figure 7(a) shows the results of an experiment in which all processes perform a *lockall–put–unlockall* pattern to all the other processes, but only the first rank of each node receives an increasing number of PUT operations (varied on the x-axis of the graph), while the others receive only one PUT operation. Our random load balancing simply chooses the ghost processes in the order of its local rank for each target process. Thus, all the PUT operations are always equally distributed to the ghosts on each node achieving much better performance that with static binding.

Figure 7(b) uses a variant of the previous experiment in which each process performs an uneven *lockall–accumulate–put–unlockall* pattern to all other processes. In this case, random load balancing arbitrarily picks the ghost process for each PUT operation but sends all ACCUMULATE operations to the same ghost process (in order to maintain ordering and atomicity guarantees). Thus, the ghost process that is handling both ACCUMULATE and PUT operations would end up having to handle more operations than would the other ghost processes. Our "operation-counting" approach, on the other hand, keeps track of which ghost process has been issued how many operations and balances the operations appropriately, thus allowing it to achieve better performance than the random approach does.



(a) Accumulate on Cray XC30. (b) Put on Cray XC30. (c) Accumulate on Fusion using MVAPICH.

Figure 5. Asynchronous progress on different platforms.



(a) Static Rank Binding: Increasing Processes. (b) Static Rank Binding: Increasing operations. (c) Static Segment Binding: Uneven Window Size

Figure 6. Load balancing in static binding on Cray XC30.

Our third experiment, uses yet another variant of the previous experiments by varying the size of the operations while keeping the number of operations constant. Each process performs a *lockall–accumulate–put–unlockall* pattern, but only the processes whose local rank is 0 receive increasing sizes of PUTs and ACCUMULATEs (varied on the x-axis), while the others receive only one double PUT and accumulate. Figure 7(c) shows the results. As expected, neither random nor operation-counting algorithms can handle this case well, although our "byte-counting" approach outperforms both of them.

D. NWChem Quantum Chemistry Application

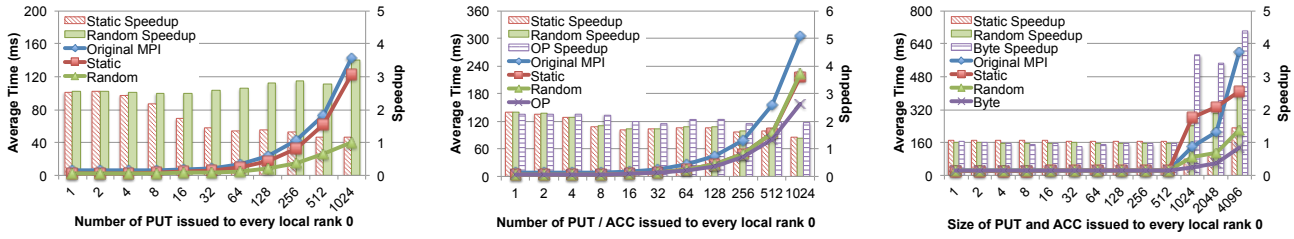
NWChem [16] is a computational chemistry application suite offering many simulation capabilities. For massively parallel simulations, a common method employed is coupled-cluster theory (CC), for which NWChem has extensive functionality [17] and excellent performance [18]. For data movement, NWChem uses the Global Arrays [19] toolkit, which has been implemented on a number of platforms natively and as a portable implementation over MPI RMA [2].

Because CC simulations are one of the most common usages of NWChem in the context of large clusters or supercomputers, our experiments focus on the most popular CC method, CCSD(T). Two molecules are considered: the water cluster $(H_2O)_n$ ($n = 16$)—denoted Wn for short—and C_{20}, obtained from the NWChem QA test suite (`QA/tests/tce_c20_triplet`). For the water cluster, we used double-zeta basis sets (cc-pVDZ from the NWChem basis set library), which are reasonable for this class of problems. We compared Casper with both the original MPI and two thread-based approaches. The first approach employs oversubscribed cores (Thread (O)), where every thread and its MPI process execute on the same core; the second approach uses dedicated cores (Thread (D)), where threads and MPI processes are on separate cores. We used the same total number of cores in all approaches, some of which are dedicated to asynchronous ghost processes/threads as listed in Table I.

Table I
CORE DEPLOYMENT IN NWCHEM EVALUATION ON CRAY XC30.

	Computing Cores	Async Cores
Original MPI	24	0
Casper	20	4
Thread (O)	24	24
Thread (D)	12	12

Figures 8(a) and 8(b) report timings for a single iteration of CCSD, which is a communication-intensive solver composed of more than a dozen tensor contractions of varying size. For smaller problems, when computation dominates, asynchronous progress is more important because the application is calling MPI relatively infrequently. At larger scale, the computation time decreases, and the communication is more frequent; hence the improvement with Casper is less. The (T) portion of the CCSD(T) methods is much more compute-intensive. Hence, the time between MPI calls can be large, and thus the impact of asynchronous progress is significant. Each process fetches remote data, then does significant computation—over and over. As a result, the lack of progress causes processes to stall, waiting on GET



(a) Random: Uneven Number of Put. (b) OP-counting: Uneven Number of Put/ACC. (c) Byte-counting: Uneven Size of Put/ACC.

Figure 7. Dynamic load balancing on Cray XC30.

operations to be satisfied remotely. Figure 8(c) shows the significance of asynchronous progress at all scales. Relative to the original version, Casper is almost twice as fast; but thread-based asynchronous progress is far less effective. The reason is that although both approaches improve asynchronous progress in communication, the thread-based solutions significantly degrade the performance of computation, either by core oversubscription or by appropriation of half of the computing cores.

V. Related Work

Asynchronous progress in MPI has been previously explored by the community for both two-sided and one-sided communication using multiple approaches. Sur et al. [20] discussed an interrupt-based design to overlap remote direct memory access read-based rendezvous communication with computation on InfiniBand networks. Kumar et al. [21] improved this work by proposing a signal-based approach to both reduce the number of interrupts and avoid using locks for shared data.

In one-sided communication, although networks such as InfiniBand provide contiguous PUT/GET operations in hardware, noncontiguous data transfers and accumulate operations still require the participation of the target process to perform an unpacking stage from a contiguous receiving buffer into the noncontiguous target location. Jiang et al. [3] proposed a thread-based design to enable asynchronous progress in communications involving noncontiguous data types in one-sided networks. Vaidyanathan et al. [22] improved asynchronous progress on Intel Xeon Phi coprocessors using a similar approach but were able to minimize threading overhead by implementing only a subset of the MPI standard and discarding some requirements of the standard.

PIOMan [23], a multithreaded communication progression engine supporting asynchronous progress, divides I/O communication and rendezvous handshakes into multiple tasks and offloads them to background threads running on idle cores in order to overlap communication and computation. This approach, however, suffers from a nonnegligible overhead derived from the necessary multithreading safety mechanisms [24]. In addition, to the best of our knowledge, the PIOMan project does not target one-sided communications.

Other research has focused on improving communication overlap using network hardware features. Santhanaraman et al. [25] optimized internode one-sided passive-mode synchronization using InfiniBand atomic operations, thus providing applications with improved overlap. Realizing that intranode communication is highly processor demanding, Zounmevo and Afsahi [26] proposed to overlap intra and internode one-sided communications by deferring the former messages falling under a certain message size threshold to the end of the epoch. By issuing network transfers to RDMA-assisted networks in first place, the processor-expensive intranode data movements are can be overlapped when issuing them subsequently.

VI. Concluding Remarks

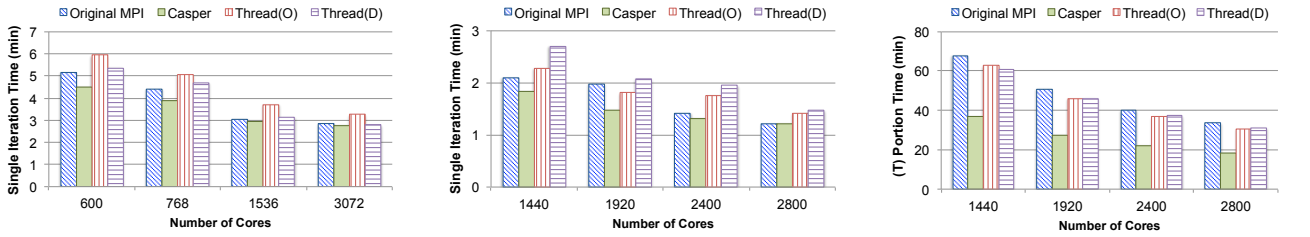
RMA communication allows a process to access memory regions of other processes without the target process explicitly needing to receive or process the message. However, the MPI standard does not guarantee that such communication is asynchronous; and some MPI implementations still require the remote target to make MPI calls in order to ensure progress on incoming RMA messages. This paper presented a process-based asynchronous progress approach in which background ghost processes are employed to assist RMA operations that require software intervention for progress on the target side without affecting hardware-based RMA operations.

Acknowledgments

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resources for this paper were provided by the National Energy Research Scientific Computing Center (NERSC) on the Edison Cray XC30 supercomputer and by the Laboratory Computing Resource Center on the Fusion cluster at Argonne National Laboratory.

References

[1] "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sep. 2012.



(a) CCSD for W16=$(H_2O)_{16}$ with pVDZ (b) CCSD for C_{20} with pVTZ (c) (T) portion of CCSD(T) for C_{20} with pVTZ

Figure 8. NWChem TCE coupled cluster methods on Cray XC30.

[2] J. S. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," in *IPDPS*, May 2012.

[3] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp, "Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters," in *Euro PVM/MPI*, ser. Lecture Notes in Computer Science, 2004, vol. 3241, pp. 68–76.

[4] Argonne National Laboratory, "MPICH — High-Performance Portable MPI," http://www.mpich.org, 2014.

[5] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu, 2014.

[6] Intel Corporation, "Intel MPI library," http://software.intel.com/en-us/intel-mpi-library, 2014.

[7] S. Kumar and M. Blocksome, "Scalable MPI-3.0 RMA on the Blue Gene/Q supercomputer," in *Euro MPI*, 2014.

[8] W. Gropp and R. Thakur, "Thread-safety in an MPI Implementation: Requirements and Analysis," *Parallel Comput.*, vol. 33, no. 9, pp. 595–604, Sep. 2007.

[9] Cray Inc., "Cray Message Passing Toolkit," http://docs.cray.com/books/S-3689-24, Cray Inc., Tech. Rep., 2004.

[10] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM, Jun. 2013.

[11] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote Memory Access Programming in MPI-3," Argonne National Laboratory, Tech. Rep., 2013.

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

[13] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI Altix 3000 Global Shared-Memory Architecture," Silicon Graphics, Inc, White paper, Apr. 2003.

[14] R. Brightwell, K. Pedretti, and T. Hudson, "SMARTMAP: Operating System Support for Efficient Data Sharing among Processes on a Multi-Core Processor," in *SC*. IEEE, 2008.

[15] C. SPARC International, Inc., *The SPARC Architecture Manual (Version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[16] E. J. Bylaska et. al., "NWChem, A Computational Chemistry Package for Parallel Computers, Version 6.3," 2013.

[17] S. Hirata, "Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories," *J. Phys. Chem. A*, vol. 107, pp. 9887–9897, 2003.

[18] E. Aprà et al., "Liquid Water: Obtaining the Right Answer for the Right Reasons," in *SC*, 2009.

[19] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers," in *ACM/IEEE conference on Supercomputing*, 1994.

[20] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA Read based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits," in *PPoPP*, 2006, pp. 32–39.

[21] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda, "Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication," in *Euro PVM/MPI*, ser. Lecture Notes in Computer Science, 2008, vol. 5205, pp. 185–193.

[22] K. Vaidyanathan, K. Pamnany, D. D. Kalamkar, A. Heinecke, M. Smelyanskiy, J. Park, D. Kim, A. Shet, G, B. Kaul, B. Joo, and P. Dubey, "Improving Communication Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters," in *IPDPS*, 2014.

[23] F. Trahay and A. Denis, "A Scalable and Generic Task Scheduling System for Communication Libraries," in *IEEE Cluster*, Sep. 2009.

[24] F. Trahay, É. Brunet, and A. Denis, "An Analysis of the Impact of Multi-Threading on Communication Performance," in *9th Workshop on Communication Architecture for Clusters (CAC)*, May 2009.

[25] G. Santhanaraman, S. Narravula, and D. K. Panda, "Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap." in *IPDPS*, 2008.

[26] J. A. Zounmevo and A. Afsahi, "Intra-Epoch Message Scheduling to Exploit Unused or Residual Overlapping Potential," in *Euro MPI*, 2014.