Scalability Challenges in Current MPI One-Sided Implementations

Xin Zhao,* Pavan Balaji,† and William Gropp*

*University of Illinois at Urbana-Champaign, {xinzhao3,wgropp}@illinois.edu
†Argonne National Laboratory, balaji@anl.gov

***Abstract*—MPI one-sided or remote memory access (RMA) communication provides a different execution model from traditional two-sided or group communication and is better suited for some classes of applications. However, current implementations of MPI RMA are notorious for their inability to scale to large systems or problem sizes. In this paper, we present a study of the RMA infrastructure in popular open-source MPI implementations. Our objective is to identify critical scalability limitations with respect to memory usage in these implementations. We then perform a thorough evaluation on two cluster computers to demonstrate those scalability limitations, and we provide suggestions on how they can be alleviated.**

I. Introduction

The Message Passing Interface (MPI) [1] is the most prominent model for parallel programming of scientific computing applications on large parallel and distributed computing systems. In MPI-2, the MPI Forum introduced one-sided or remote memory access (RMA) communication. In MPI-3, the RMA capabilities of MPI were significantly revised, leading to newer, cleaner, and more performance-capable RMA semantics. In the RMA model, one process can directly access the memory on another process without requiring explicit communication calls from the target. This model provides a different execution paradigm from traditional two-sided or group communication, offering an attractive alternative for some applications.

Several researchers have investigated MPI RMA as an alternative to their existing usage of MPI, particularly for applications with irregular communication patterns. Such studies are motivated by the fact that MPI RMA does not require processes to be "cooperative"; in other words, the origin does not need to explicitly match a call on the target. This approach improves the ease of writing applications especially when communication is irregular and data driven, because the programmer no longer needs to carefully plan the communication pattern. For example, in NWChem [2], a large quantum-chemistry application, massive asynchronous messages need to be communicated; and a process typically does not know from whom to receive the message. Thus, it uses RMA to implement a model where each process can dynamically fetch data, perform the computation, and write the computed output to a target remote memory region.

Despite its growing prominence, however, current implementations of MPI RMA are notorious for their inability to scale to large systems or problem sizes. A commonly mentioned example in the literature is the failure of MPI RMA to scale with the Graph 500 benchmark [3]. Specifically, most MPI implementations run out of internal resources when scaling Graph 500 to large problem or system sizes. Irregular communication, such as that evidenced in Graph 500, involves a large number of outgoing asynchronous operations; and the MPI implementation might need to maintain state for each of them. This situation can cause the MPI implementation to consume large amounts of memory, leaving none for the application. Similarly, since each process can communicate with many other peers, the MPI implementation might have to maintain state for all possible communication peers, a situation that can cause $O(P)$ memory consumption, where P is the number of MPI processes in the system.

In this paper, we study four open-source MPI RMA implementations—MPICH, MVAPICH, Open MPI, and foMPI—examining their strengths and shortcomings and demonstrating the different strategies they employ within their MPI RMA infrastructure. The primary objective of this work is to identify the scalability challenges in current MPI implementations and propose what needs to change in order to efficiently support MPI-based applications on extreme-scale systems. The purpose of this study is to analyze the conceptual features provided by the different implementations, rather than the software engineering of these different features. Therefore, we implemented all the MPI RMA features from each of these open-source implementations into a common code base, thus allowing for an apples-to-apples comparison of the features. Together with a detailed analysis of the various features provided by the various implementations, we also present a thorough performance evaluation and analysis on up to 2,048 MPI processes.

Prerequisites. This paper assumes that the reader is familiar with the semantics of MPI RMA. We recommend that those unfamiliar with MPI RMA first read through past papers and books ([4], [5], [6]) on the topic. We also note that we do not include an explicit "related work" section. Instead, we describe the various existing MPI implementation capabilities where appropriate in the text, and we cite the appropriate

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DEAC0206CH11357.

references as needed.

II. Overview of RDMA on Modern Networks

Before we discuss the RMA infrastructure in current MPI implementations, we need to first clarify how hardware networks work. In this section, we present the relevant details for common supercomputing networks with an emphasis on the features that impact MPI implementations with respect to performance or scalability.

RMA, typically referred to as remote direct memory access (RDMA) for network hardware, is common on most networks today. However, each network provides these capabilities in a slightly different manner. Thus, to understand the design choices of MPI RMA implementations, we must first understand the subtle differences in the RDMA capabilities of various networks. In this section, we present a survey of RDMA capabilities on six different networks: Mellanox InfiniBand [7], [8], Portals-4 on Bull BXI [9], [10], the Tofu network architecture [11], the Cray Aries [12], the IBM Blue Gene/Q network (BG/Q) [13], and RDMA over Converged Ethernet (RoCE) [14].

For Mellanox InfiniBand, our survey is based on the ConnectX-4 hardware. Portals-4 is technically only an API, not a hardware specification, and currently no hardware implements it; however, several network hardware implementations of Portals-4 have been announced, such as the BXI interconnect from Bull. Our survey is based on the available open literature on BXI. The Tofu network architecture is based on Fujitsu's K computer and the follow-on Fujitsu FX100 computer [15]; these are two separate generations of the Tofu network, but for the high-level overview provided in this section, they are indistinguishable. For Cray Aries, our survey is based on the Cray XC30 system. BG/Q is the third generation in the IBM Blue Gene line of massively parallel supercomputers. RoCE is a network protocol that allows RDMA over Ethernet networks; our survey is based on the RoCE version 2 protocol.

Table I: RDMA capabilities provided by modern networks

Network	Window Address	Protection Keys
InfiniBand	HW-addr	Yes
Portals4	HW-offset	No
Tofu	HW-addr	No
Cray Aries	HW-addr	No
IBM BG/Q	HW-offset	No
RoCE	HW-addr	Yes

Of the various hardware capabilities provided by these networks, two capabilities are of particular interest to us with respect to their influence on the various designs of MPI RMA: (1) window address calculation and (2) memory protection keys. Table I summarizes those capabilities on the six different networks in our survey. In the following subsections, we provide more details on these two capabilities.

A. Window Address Calculation

Different networks provide different ways of referring to the buffer to which an RDMA operation needs to be targeted. For example, networks such as Mellanox InfiniBand require that the MPI implementation provide the absolute virtual address of the memory location on the target—we refer to this model as "HW-addr"-based RDMA. Networks such as Portals-4, on the other hand, allow the implementation to access the target buffer using an offset from a base memory location—we refer to this model as "HW-offset"-based RDMA. MPI RMA uses a slightly different model from these two models. In the MPI RMA model, the user specifies the target memory location as a scaled offset. That is, the target memory region is identified by using an array offset, where each element of the array is of a predefined size (e.g., array of integers or array of doubles). Thus any RMA communication needs to be scaled by the appropriate array element size before any communication is performed. We refer to this model as "scaled offset"-based RDMA.

This mismatch in the semantics of MPI RMA and the semantics of the network RDMA capabilities has subtle implications for the memory usage of the various MPI implementations. The MPI implementation needs to translate the user-specified "scaled offset"-based operations to the appropriate RDMA model expected by the network hardware. Thus, it needs to maintain appropriate metadata in order to handle such translation. For example, on networks that provide "HW-addr"-based RDMA, the MPI implementation needs to maintain an $O(P)$ array of base or start addresses and another $O(P)$ array of scaling units in order to do this translation. For networks that provide "HW-offset"-based RDMA, on the other hand, one no longer has to maintain an $O(P)$ data structure for the base addresses but does need to maintain such a structure for the scaling units. Fortunately, in practice, most applications provide the same scaling unit on all processes, thus requiring constant memory. At the time of writing this paper, no publicly announced network provides "scaled offset"-based RDMA.

B. Memory Protection Keys

RDMA allows multiple processes to directly access a target's memory. While convenient, it is also a security concern, particularly in non-scientific computing environments where multiple users might share the same node or in secure environments where multiple users share the same supercomputer and protection between users is desired. To accommodate such cases, some networks use a memory protection key where only processes that have this protection key can access the target's memory. Mellanox InfiniBand is an example of such a network, where a memory protection key, called an "rkey," is required in order to access a memory location with RDMA. This key is asymmetric. That is, each process can potentially generate a different key, thus requiring the origin to store a key for each potential target: an $O(P)$

memory usage. Other networks, such as Tofu, require no keys to access memory. While scalable, the model used by Tofu requires that security against unauthorized accesses be managed separately.

III. Scalability Issues in MPI Implementations

MPI implementations have several scalability limitations in their RMA infrastructure. Some of these limitations are common to all implementations, while others exist only in specific implementations. In this section, we present the various implementation choices for RMA that are used in four open-source MPI implementations: MPICH (3.1.4) [16], Open MPI (1.10.2) [17], MVAPICH (2.2b) [18], and foMPI (0.2.1) [19]. Table II summarizes the scalability challenges in these implementations.

A. Implementation Choices for MPI RMA Operations

While conceptually MPI RMA is similar to network hardware RDMA, they are not exactly alike. Network hardware RDMA is often restrictive in the capabilities it provides. For example, most networks provide simple RDMA communication (such as writing (PUT) or reading (GET) of contiguous data) in hardware. But more complex communication, such as ACCUMULATE or RMA operations with noncontiguous data segments, is often not directly implemented in hardware. MPI RMA, on the other hand, is more general, providing users with a wide variety of features, not all of which are provided by network hardware RDMA. Consequently, some MPI RMA operations map directly (or "natively") to network hardware RDMA, while other MPI RMA operations do not have a native equivalent and thus need to be emulated.

The general approach used by most MPI implementations to handle this issue is to use hybrid communication based on network-hardware-based (HW-based) operations and active-message-based (AM-based) operations [20]. AM-based operations are software-emulated RDMA operations, where the origin process sends a message and a software handler (also known as the active-message handler) is triggered on the target side. This active-message handler then performs the appropriate action at the target process and returns any required data to the origin.

RMA between processes on the same node is often implemented by direct memory accesses to a shared-memory region. We treat this as HW-based RMA in this paper.

Limitations of foMPI. HW-based operations typically achieve higher performance, but AM-based operations are more general. Consequently, most MPI implementations, including MPICH, Open MPI, and MVAPICH, use a hybrid model that chooses either HW-based or AM-based communication depending on the operation. An exception is foMPI, however, where all RMA operations are implemented by using HW-based operations with the intention of taking advantage of their better performance. This approach, however, comes at a cost. Operations that do not have a native hardware RDMA equivalent need to be emulated by using multiple hardware RDMA operations. Doing so causes multiple network communication operations, leading to performance loss. As we will show in Section IV, sometimes this performance loss is so severe that the model results in more than an order-of-magnitude worse performance compared with that of AM-based operations.

An example of such degradation for ACCUMULATE operations is presented in Figure 1. As discussed above, most networks provides native hardware RDMA for contiguous PUT/GET communication but not for atomic ACCUMULATE operations such as summation or bitwise AND/OR. For such operations, foMPI uses the following protocol. It first performs hardware COMPARE_AND_SWAP over the network in a loop waiting for a window mutex on the target to be acquired. This mutex is required in order to maintain the atomicity of ACCUMULATE operations. Once the mutex is acquired, the origin issues a HW-based GET to fetch the target data and performs the computation locally. After the computation is finished, the origin issues a HW-based PUT to push the results back to target. The origin then issues a HW-based COMPARE_AND_SWAP to release the window mutex. This method incurs significant overhead because each RMA operation, which would have been a single one-way transaction with AM-based communication, has now translated into six one-way network transactions (note that we do not wait for the return value in the second COMPARE_AND_SWAP). If multiple atomic operations are competing for the same memory location, the number of network transactions can be more than six.

Another example of such degradation is with RMA operations for noncontiguous data. In this case, foMPI analyzes derived datatypes and translates them into multiple small contiguous RDMA operations over the network. Such a strategy allows the MPI implementation to rely on native hardware RDMA operations but incurs significant overhead when the number of noncontiguous segments is large. As we will discuss in Section IV, this can degrade performance by more than an order of magnitude compared with AM-based operations.

Limitations of MPICH, MVAPICH, and Open MPI. MPICH and Open MPI are portable to more networks than MVAPICH and foMPI are. However, not all their features are portable to all network configurations. MPICH provides direct RMA operations only for shared memory and relies on AM-based RMA operations for other networks (e.g., InfiniBand and Portals-4). Open MPI relies on AM-based RMA operations for most networks (e.g., InfiniBand); for Portals-4, however, it uses a hybrid approach utilizing both HW-based and AM-based RMA operations. MVAPICH is implemented only for InfiniBand and uses a hybrid approach utilizing both HW-based and AM-based RMA operations, similar to what Open MPI does for Portals-4.

Table II: Scalability challenges in existing MPI one-sided implementations

Aspect	MPICH	MVAPICH	Open MPI	foMPI
Window metadata storage	$O(P)$ memory usage			
Synchronization in WIN_FENCE	REDUCE_SCATTER synchronization ($O(P)$ memory) for AM-based implementation and BARRIER synchronization for HW-based implementation			BARRIER synchronization
Metadata for targets	$O(P)$ memory usage for AM-based implementation			Constant
Managing concurrent passive locks	Queued locks ($O(P)$ memory)		Lock polling for HW-based implementation and queued locks ($O(P)$ memory) for AM-based implementation	Lock polling
Operation issuing strategies	Eager issuing for HW-based implementation and delayed issuing (unlimited memory usage) for AM-based implementation			Eager issuing
Metadata for outgoing operations	Unlimited memory usage			Constant

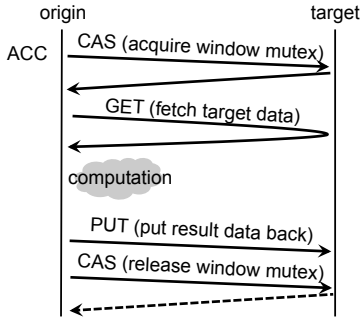


Figure 1: Implementing ACCUMULATE using network-hardware-based operations

B. Window Metadata Storage

In MPI, before any RMA operation is issued, each process must declare a part of its memory as "remotely accessible." This part is referred to as "window creation." MPI provides four ways of creating a window: WIN_CREATE, WIN_ALLOCATE, WIN_CREATE_DYNAMIC, and WIN_ALLOCATE_SHARED. Window creation is collective. During this phase, the MPI implementation can exchange the necessary metadata between different processes, allowing them to access each other's memory. This metadata includes user-specified information such as the start address of the target buffer on the window and the size of the scaling unit, as well as network-hardware-specific metadata such as memory protection keys.

The kind of metadata that needs to be stored by the MPI implementation depends on the capabilities provided by the network hardware as well as the kind of RMA operations that the application uses, specifically operations that are HW-based or AM-based. During window creation time, the MPI implementation has information about the network hardware but not about the kind of RMA operations that would be issued by the application. Thus, it has to maintain the metadata required for both AM-based and HW-based operations. We will discuss the metadata required for these two kinds of operations separately.

1) AM-Based Operations: AM-based operations are generic and customizable. When an MPI RMA operation is implemented by using AM-based operations, any information required for accessing the target memory location, such as the scaling unit, can be directly accessed at the target. Thus, the origin process does not need to maintain such metadata. Some MPI implementations, for example Open MPI, implement AM-based operations this way, thus using $O(1)$ memory for the associated metadata. MPICH and MVAPICH, on the other hand, store the window base address for each target process, even for AM-based communication, thus using $O(P)$ memory for the associated metadata. This approach is both unnecessary and wasteful. As described in Section III-A, foMPI does not support AM-based operations.

2) HW-Based Operations: HW-based operations are, in general, more efficient and performant than AM-based operations are. However, such performance benefit comes at the cost of additional metadata storage requirements.

Metadata for Memory Protection Information. On networks that require memory protection keys, each process has to store this information for every target buffer. This process requires $O(P)$ memory. An example of such behavior is MVAPICH, which stores this metadata for the InfiniBand network. MPICH and Open MPI support InfiniBand but do not provide HW-based RMA for it and hence do not need to maintain such metadata. The foMPI implementation does not support InfiniBand. For networks that do not require memory protection keys, on the other hand, no such metadata storage is required. An example is Open MPI over Portals-4. MPICH supports Portals-4 but does not provide HW-based RMA for it. MVAPICH and foMPI do not support Portals-4.

Metadata for Window Base Addresses. For networks that require an absolute virtual address on the target for RDMA operations, the origin needs to store the base addresses for each target process, in the general case. This process requires $O(P)$ memory. An example of such behavior is MVAPICH, which stores this metadata for InfiniBand when the base addresses are not symmetric. When a window is created with WIN_ALLOCATE, the MPI implementation can use a technique called "symmetric allocation" to allocate the same base address on all processes, so that each process consumes only $O(1)$ memory. This strategy is used by foMPI.

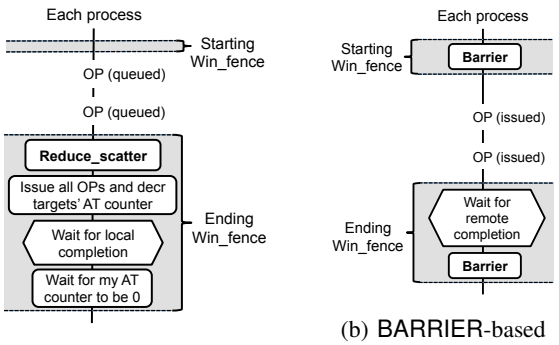
Metadata for Window Scaling Unit. At the time of writing this paper, no publicly announced network directly provides "scaled offset"-based RDMA. Thus, in cases where the user creates a window with different scaling units for each process, the MPI implementation must maintain the necessary metadata to store this information. MPICH, MVAPICH, Open MPI, and foMPI all consume $O(P)$ memory in this case.

C. Epoch Synchronization

All MPI RMA operations must be enclosed within an epoch. The origin needs to ensure that the target is "ready" before it can issue RMA operations to that target. Two models for epoch synchronization exist. The first model is called "active target synchronization," where the origin and target explicitly synchronize before RMA operations can be issued. One example of this model is WIN_FENCE. The second model is called "passive target synchronization," where the target is not explicitly involved in the synchronization, so the origin needs to implicitly coordinate with other potential origins before issuing RMA operations to that target. One example of this model is WIN_LOCK. In this section, we discuss the metadata requirements for both models.

1) Fence Epoch: WIN_FENCE is a common epoch model in MPI RMA, used by applications such as Graph 500. In this model, all processes open an epoch by calling WIN_FENCE, issue RMA operations to each other, and close the epoch by calling WIN_FENCE again. An origin can issue RMA operations intended for a target only after the target has called its epoch-opening WIN_FENCE. The return of the epoch-closing WIN_FENCE on a process guarantees that (1) all operations that it has issued have locally completed and (2) all operations targeting it are now visible to that process. The first guarantee is particularly important for performance; any approach that forces a stronger guarantee, such as remote completion for all operations, would be at a performance disadvantage.

In the MPI implementations that we surveyed, two algorithms are used for WIN_FENCE: REDUCE_SCATTER-based (RS-based) and BARRIER-based (Figure 2).



(a) RS-based algorithm

(b) BARRIER-based algorithm

Figure 2: Fence algorithms

In the RS-based algorithm (Figure 2a), the epoch-opening WIN_FENCE is a no-op; all posted operations are buffered by the MPI implementation. The epoch-ending WIN_FENCE first performs a REDUCE_SCATTER, so each target knows the number of origins that will issue operations to it, and initializes a local counter (AT) with that value. Next it issues all the buffered operations, in which the last operation to each target decrements the target's AT counter. It then waits for the local completion of all the operations that it issued and for its AT counter to become zero, in other words, that all operations targeting it have been completed. This algorithm does not have additional synchronization other than that required by the MPI standard, and thus it is ideally suited for performance. However, it has two limitations. First, REDUCE_SCATTER requires $O(P)$ memory. Second, this algorithm assumes that a process knows when it is targeted by an RMA operation—an assumption that is not true for several networks (e.g., InfiniBand) when using hardware RDMA. Consequently, it can be used only for implementations where all operations are AM-based or for implementations over networks that provide remote notification semantics for RDMA (e.g., Fujitsu Tofu). MPICH and Open MPI use this algorithm for networks where no HW-based operations are supported, for example, MPICH for non-shared-memory networks and Open MPI for all networks other than Portals-4.

In the BARRIER-based algorithm (Figure 2b), the epoch-opening WIN_FENCE performs a BARRIER among all processes on the window. The epoch-closing WIN_FENCE waits for remote completion for all the issued operations and then performs another BARRIER. This algorithm is scalable with respect to the number of processes in the system, since the memory it uses is independent of the system size. However, it imposes a stricter synchronization than what the MPI standard requires. In particular, it forces the epoch-opening WIN_FENCE to synchronize between all processes—which is not required by the MPI standard. Moreover, the origin processes have to wait for remote completion of all operations at the epoch-closing WIN_FENCE—which is stricter than the local completion required by the MPI standard. Thus, while this approach is scalable with respect to memory usage, it can lose performance compared with the RS-based algorithm. MPICH, Open MPI, and foMPI use this algorithm for HW-based operations. MVAPICH uses both the BARRIER-based and RS-based algorithms simultaneously within WIN_FENCE.

2) WIN_LOCK Epoch: Like WIN_FENCE, WIN_LOCK is also a common epoch model and is used by applications such as NWChem [2]. When using WIN_LOCK, two aspects can cause the MPI implementation to consume memory that scales linearly with the number of processes in the system. We discuss both aspects here.

Target Objects. With WIN_LOCK, the MPI implementation needs to maintain a "target object" for each target in order

to maintain certain state information, thus requiring $O(P)$ memory storage. Three types of state information exist: user hints, an error-checking flag, and a lock-state flag.

User hints are recommendations given by the application for cases where the application does not need the full generality offered by the MPI standard, thus allowing the MPI implementation to optimize performance or resource usage. One example of such user hints is MODE_NOCHECK. This hint implies that the application would algorithmically ensure that there will not be any conflicting locks to the same target on the same window. The MPI implementation can use this hint to optimize performance by not acquiring a lock. If it does so, however, it needs to ensure that it also will not unlock the target, since no lock has been acquired.

Technically, checking for errors is "best effort" as defined in the MPI standard. However, most MPI implementations tend to provide some error checking, primarily as a convenience to users. One example of such error checking is for the lock state. Specifically, the MPI standard prohibits one origin from locking the same target more than once in a nested fashion. Maintaining this information within the target object allows the MPI implementation to detect and report such errors. Doing so, however, requires the MPI implementation to store this information on a per-target manner, typically in an error-checking flag.

The lock-state flag indicates the status of the lock acquisition. Two approaches are used to acquire locks. One approach is based on a synchronous lock: the WIN_LOCK call does not return until the lock at the target is acquired. This is a stricter definition than what is required by the MPI standard; and although it is correct, it can degrade performance. A second approach is based on an asynchronous lock: the WIN_LOCK call initiates the lock acquisition but returns before the lock is acquired. This approach is often more efficient because the lock acquisition can be overlapped with other computation at the origin. If the user issues RMA operations before the lock is acquired, however, the MPI implementation must buffer such operations because they cannot be issued to the target before the lock is acquired. The state of whether a lock has been acquired or not for each target is stored in the lock-state flag. Because of the pros and cons of the two approaches, MPICH, MVAPICH, and Open MPI provide hybrid models utilizing both the synchronous and asynchronous approaches. However, foMPI provides only the synchronous approach. The lock state flag is needed only in the asynchronous approach.

Lock Queuing. In the WIN_LOCK epoch, the target is not involved in the opening and closing of the epoch; instead, the origin has to coordinate with other potential origins before it can access the target. The general model to do this coordination is to have an access mutex at the target that maintains "ownership" with respect to which origin has access to the memory on that target. Concurrent locks to the same target might get serialized while trying to acquire such ownership.

Two strategies are used in MPI implementations to manage such ownership. Open MPI (for Portals-4) and foMPI use a "lock polling" strategy. In this strategy, WIN_LOCK is a synchronous lock where the origin attempts to acquire the lock by repeatedly issuing lock-attempt messages to the target until the lock is granted. This strategy, unfortunately, generates significant network traffic during WIN_LOCK, and it cannot guarantee starvation freedom among competitors. On the other hand, MPICH, MVAPICH, and Open MPI (for InfiniBand) use a "queued locks" strategy. In this strategy, WIN_LOCK issues a lock query to the target. The target grants the lock to the first query and buffers all other queries in a priority queue. When the current lock is released, the target grants the lock to the next query in that queue. This strategy involves no additional network traffic because each competitor issues only one message to the target. It also guarantees starvation freedom among competitors. The disadvantage of this approach, however, is that the priority queue can potentially have a query from each process in the system, thus consuming $O(P)$ memory in the worst case.

D. Data Movement Operations

Apart from the storage required for the window and target metadata, communication operations in MPI RMA themselves require additional metadata. Before we discuss the details of such metadata, however, we first explain how the issuing of RMA operations works.

Two strategies are used to issue RMA operations in MPI implementations: *eager-issuing* and *delayed-issuing* strategies [21]. In the eager-issuing strategy, the epoch-opening call is a blocking call; it completes the synchronization (e.g., BARRIER in Figure 2b) before the call returns. In this model, the origin can issue RMA operations as soon as they are posted by the application. The downside, however, is that the epoch-opening strategy is synchronous and thus hurts performance. In the delayed-issuing strategy, the epoch-opening call is essentially a no-op, and the synchronization is delayed to the epoch-closing call. This model requires that the MPI implementation buffer the metadata for all posted operations to be later issued during the epoch-closing call. By doing so, synchronization in the epoch-opening call can be avoided.

1) Operation Objects: Every MPI RMA operation is associated with some metadata, such as the buffer address, data count, datatype, and computation type (in ACCUMULATE-like operations). If the MPI implementation issues each operation as soon the operation is posted, it does not have to maintain this metadata. If, on the other hand, the MPI implementation chooses to buffer the operation for later issuing, it has to maintain this metadata in an "operation object."

In the delayed-issuing strategy, all operations need to be buffered until the synchronization with the target is completed. Each buffered operation allocates an operation object that stores the required metadata for that operation. This strategy improves performance because synchronization in the epoch-opening call can be avoided. Such performance improvement, however, comes at the cost of additional memory usage. In general, as the number of processes involved in the synchronization increases, the time required for the synchronization increases. Thus, for larger systems, we can expect synchronizations that involve all processes (such as WIN_FENCE) to take longer to complete. Consequently, more operations have to be buffered, thus using more memory and causing scalability concerns as the problem size and the number of operations grows. MPICH, MVAPICH, and Open MPI use eager issuing for HW-based operations and delayed issuing for AM-based operations, whereas foMPI uses only the eager-issuing strategy.

2) Request Objects: Once an RMA operation is issued by the MPI implementation, the metadata needed for issuing the operation (i.e., the operation object) is no longer needed and can be safely discarded. However, some metadata is still required to track the completion of the operation. For HW-based operations, such metadata is typically tracked by the network hardware, in the form of either completion events (e.g., InfiniBand) or a single counter that tracks the number of operations completed (e.g., Portals-4). For AM-based operations, however, the MPI implementation needs to keep track of the state of issued but incomplete operations in an object called "request" in order to detect their completion. MPICH, Open MPI, and MVAPICH use request objects for AM-based operations. Unfortunately, none of the MPI implementations have any resource management strategy to manage such request objects. Consequently, these objects can easily use up internal resources when there are a large number of incomplete operations. Since foMPI offloads all RMA operations to the hardware, no request objects need to be maintained.

IV. Evaluation

In this section, we evaluate the impact of the various strategies used in different MPI implementations with respect to performance and memory usage. We use two clusters for our evaluation. The first cluster, "Fusion," is configured with Mellanox InfiniBand and has 320 nodes; each node contains 8 cores and 36 GB of memory. The second cluster, "Breadboard," consists of 16 nodes and is configured with Portals-4; each node contains 16 cores and 16 GB of memory. The Portals-4 network stack is the reference implementation from Sandia National Laboratories, implemented over InfiniBand. Since our focus is on the strategies used by the different implementations, to keep the comparison fair, we ported all the strategies from all the studied MPI implementations into a single code base based on MPICH-3.1.4. All comparisons are made using this code base.

For all experiments, we started with 100,000 iterations for small scales (e.g., small message sizes, small number of processes). As we scaled up the test, we reduced the iteration count so as to keep the time taken by each subexecution between 20 and 40 seconds, which was sufficient to gain reasonable statistical confidence in our experiments. The tests themselves were executed 10 times, and the statistical error bars are shown in each figure.

A. Performance of HW-Based and AM-Based Operations

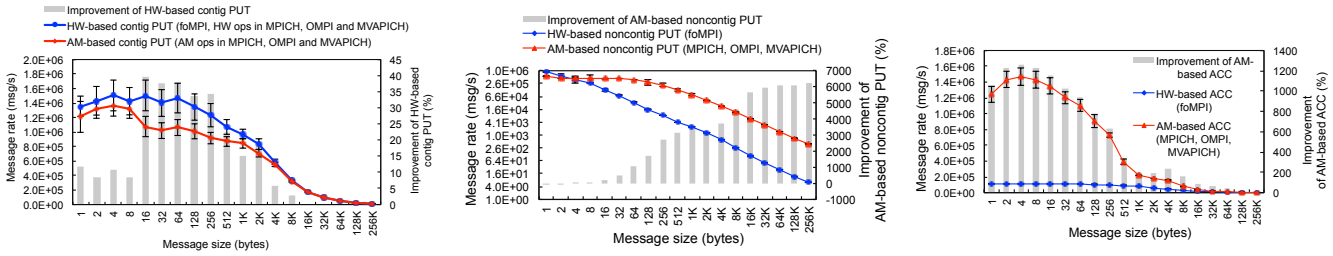
As discussed in Section III-A, MPICH, MVAPICH, and Open MPI use a hybrid model that combines both HW-based and AM-based operations; foMPI, however, converts all operations to HW-based operations, often at the cost of multiple additional HW operations. In this section, we compare the performance of the foMPI strategy with that used by MPICH, MVAPICH, and Open MPI. We compare three types of RMA operations: (1) PUT with contiguous data, (2) PUT with noncontiguous data, and (3) ACCUMULATE with a basic datatype.

As expected, HW-based PUT performs better than AM-based PUT for contiguous data by up to 40% (Figure 3a). When noncontiguous derived datatypes[1] are used, however, the performance of HW-based PUT is more than an order of magnitude worse than that of AM-based PUT. The reason is that, as described in Section III-A, the HW-based implementation translates each noncontiguous PUT into multiple small contiguous hardware PUTs. This step significantly increases the number of hardware transactions compared with the AM-based approach. HW-based ACCUMULATE is implemented as shown in Figure 1. AM-based ACCUMULATE is implemented by triggering a handler at the target that acquires a window mutex, performs computation, and releases the mutex. Our benchmark performs "ACCUMULATE + WIN_FLUSH_LOCAL" in a loop. Figure 3c shows the message rate achieved. Again, the AM-based implementation significantly outperforms the HW-based implementation because the latter needs to convert each ACCUMULATE into multiple hardware transactions.

B. Window Metadata Storage

The next aspect that we compare is the metadata storage for the window. As mentioned in Section III-B1, no metadata storage is needed when the MPI implementation uses only AM-based operations. However, MPICH, MVAPICH, and Open MPI use both HW-based and AM-based operations; and foMPI uses only HW-based operations. Thus, all MPI implementations require some metadata, as appropriate to the network they are using.

[1] vector type, where each block is a single byte and the number of blocks was increased to create larger messages; we used a stride of two bytes.



(a) Message rate of contiguous PUT

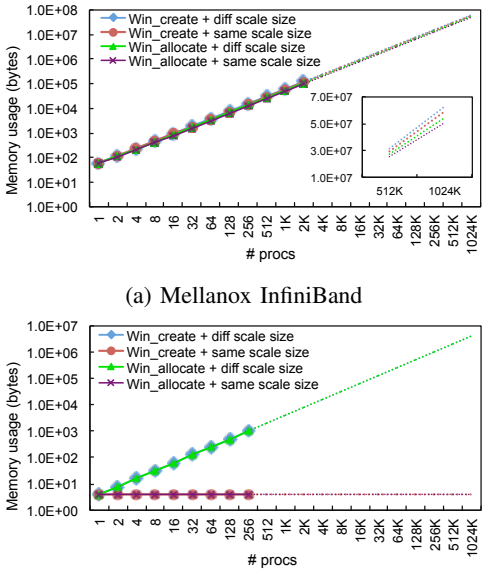
(b) Message rate of noncontiguous PUT

(c) Message rate of ACCUMULATE

Figure 3: Performance comparison of HW-based and AM-based operations (InfiniBand)

Figures 4a and 4b show the memory usage of different window creation schemes. The solid lines indicate measured memory usage (up to 2,048 processes), while the dotted lines indicate predicted memory usage on larger systems. We measured four combinations of window creation schemes: (1) WIN_CREATE with different scaling units on each process, (2) WIN_CREATE with the same scaling unit on all processes, (3) WIN_ALLOCATE with different scaling units on each process, and (4) WIN_ALLOCATE with the same scaling unit on all processes.

Figure 4a shows the memory usage on InfiniBand. Recollect that InfiniBand requires absolute virtual addresses on the target and memory protection keys for communication. Thus, three sets of metadata need to be maintained in the worst case: window base addresses (8 bytes), scaling unit sizes (4 bytes), and asymmetric remote protection keys (48 bytes). Therefore, all four schemes increase linearly with the number of processes, using up to 120 KB per process for 2,048 processes (measured) and 60 MB per process for 1 million processes (estimated). Figure 4b shows the memory usage on Portals-4. Recollect that Portals-4 uses address offsets, instead of absolute virtual addresses at the target. Thus, window base addresses do not need to be maintained. Similarly, Portals-4 does not require any memory protection keys. The only piece of metadata needed would be the scaling unit, which can be $O(P)$ if each process provides a different scaling unit size, thus using up to 1 KB per process for 256 processes (measured) and 4 MB per process for 1 million processes (estimated).



(a) Mellanox InfiniBand

(b) Portals-4

Figure 4: Memory usage of different window schemes

C. Epoch Synchronization Metadata

Next we compare the different epoch synchronization strategies presented in Section III-C.

Different Approaches for WIN_FENCE. As described in Section III-C1, two algorithms are possible for WIN_FENCE: RS-based and BARRIER-based. We performed two studies comparing these approaches. The first study was for performance, as shown in Figure 5. The BARRIER-based algorithm does additional synchronization for the epoch-opening WIN_FENCE as well as the RMA operations themselves (by waiting for their remote completion). This causes it to lose performance compared with the RS-based algorithm. In our experiment, this performance difference is 70% for 2,048 processes. The second study was for memory usage. As expected, the memory usage increases as $O(P)$ for the RS-based algorithm, while the BARRIER-based algorithm does not use any additional metadata. These results are straightforward, so we have not plotted them in the paper.

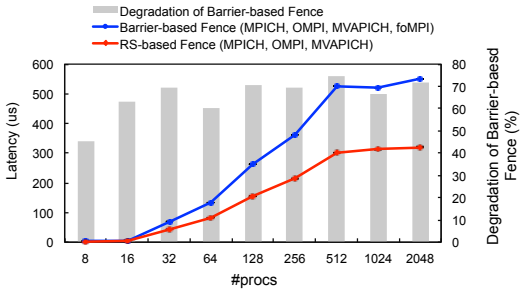


Figure 5: Comparison of fence algorithms (InfiniBand)

Management of Target Objects. We studied the memory usage for the target objects described in Section III-C2. As with the RS-based algorithm for WIN_FENCE, the memory

usage increases as $O(P)$. Again, these experimental results were straightforward, so we have not plotted them in the paper.

Lock Queuing. In Figure 6, we compare the two lock management strategies discussed in Section III-C2: "queued locks" and "lock polling." In our experiment, we use all-to-one communication where multiple origins issue an exclusive WIN_LOCK to the same target, followed by one or more 1-byte RMA operations. Figure 6a shows the performance when there is a single operation within the epoch. At 2,048 processes, queued locks perform nearly 3.5-fold better than lock polling. The reason is that lock polling generates significant network traffic, causing network contention. Figure 6b shows the performance when 64K operations are issued within the epoch. Queued locks perform better than lock polling by nearly an order of magnitude for 2,048 processes. The advantage is larger here, compared with the case where a single operation is issued within the epoch, because issuing more operations worsens the network contention irrespective of whether these operations are for lock acquisition or for actual data communication.

Figure 6c shows the memory usage for the two strategies. The memory usage of lock polling is zero as expected, while that of queued locks increases linearly with the number of origins, peaking at around 70 KB for 2,048 processes for each MPI window, in other words, 32 bytes per origin per window. Thus, on a system with a million processes, where each process creates 10 windows, the memory usage per process can be as high as 320 MB. This is, of course, the worst-case scenario. In practice, however, an application is unlikely to have an access pattern where all origins attempt to lock the same target, which is highly unscalable. Thus, the actual memory usage can be expected to be considerably lower.

D. Data Movement Operations

Here we analyze the memory usage of the operation and request objects for the two approaches described in Section III-D.

1) Total Memory Usage per Process: We first measure the total memory usage of the MPI implementation in an all-to-all communication. We restrict the number of processes to 16, distributed over two nodes, and increase the number of operations issued by each process. The results are shown in Figure 7. In the eager-issuing strategy, the memory usage is flat (this memory is used by the MPI implementation to store other metadata), whereas in the delayed-issuing strategy the memory usage increases with the number of operations posted. In fact, the amount of memory used is so high that the benchmark runs out of memory and fails to execute for more than 4 million operations.

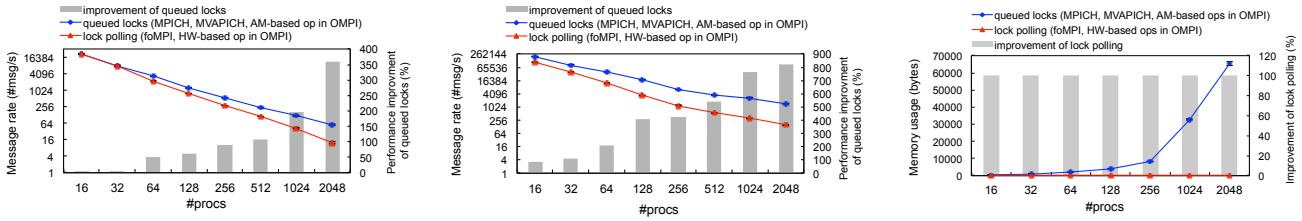
2) Metadata for Operation Objects: We next divide this memory usage into that used by the operation objects and that used by the request objects. The benchmark we used does an all-to-all communication where each origin issues 64K operations to each target. This pattern is repeated for four fence epochs. Figure 8a shows the memory usage profile for operation objects over time. Recollect that this metadata is maintained when an operation is posted by the application but has not yet been issued on the network. This behavior occurs only in the delayed-issuing strategy since the eager-issuing strategy does not buffer any operations and thus does not need to store such metadata. We note that memory usage increases at the beginning until the execution enters the epoch-ending WIN_FENCE; the memory usage then drops. The reason is that all posted operations are buffered until the epoch-ending WIN_FENCE. During the epoch-ending WIN_FENCE, all the posted operations are issued out by the MPI implementation, and memory usage decreases.

We observe that the memory usage reduction goes through three phases. In the first phase, the reduction is steep. In the second phase, some reduction still occurs, but the rate of reduction is considerably lower. In the third phase, again the reduction is steep. This last result was unexpected, so we analyzed the MPI implementations that use this strategy (MPICH and MVAPICH). We found that this behavior stems from how the garbage collection (GC) works in these MPI implementations. Specifically, the GC checks for operations that have already completed and frees them. For performance reasons, the GC does not check the entire queue every time but only a limited subset of the queue. One can expect that the first few operations to a process complete quickly while the later operations take longer to complete. However, since these MPI implementations maintain a single queue for all processes, a partial parsing of the queue will observe only a few completions, even if there are more. Thus the GC code spends more time parsing the queue and consequently loses performance. This is a software engineering issue and can be fixed by improving the design of the GC code; it should not be interpreted as an artifact of the delayed-issuing strategy.

3) Metadata for Request Objects: Figure 8b shows the profile of the memory used by the request objects for operations that have been issued but have not yet completed. We notice that the memory usage increases during the first WIN_FENCE but stays constant until the end of the application execution. The reason is that MPI implementations typically tend to minimize the amount of memory allocation and deallocation required by storing requests that are once allocated, without freeing them until FINALIZE.

V. Concluding Remarks

In this paper, we investigated four open-source MPI implementations—MPICH, MVAPICH, Open MPI, and foMPI—and identified several scalability limitations in their RMA infrastructures. The scalability limitations illustrated in this paper involve window creation, synchronization, and data movement. To demonstrate these limitations in a fair comparison, we implemented the various features from all



(a) Message Rate (#operations = 1)

(b) Message Rate (#operations = 64K)

(c) Memory Usage

Figure 6: Comparison of locking strategies (InfiniBand)

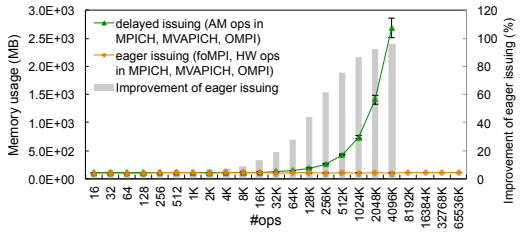
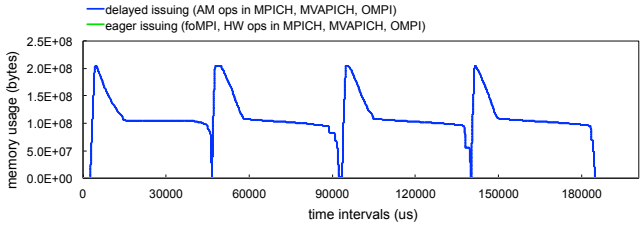
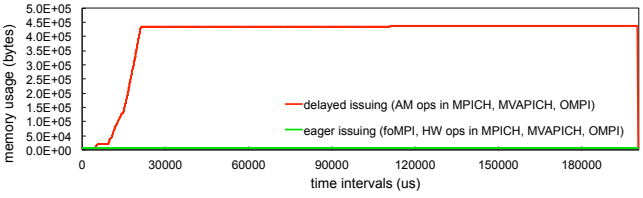


Figure 7: Memory usage in delayed issuing (InfiniBand)



(a) Memory Usage of Operation Objects (InfiniBand)



(b) Memory Usage of Request Objects (InfiniBand)

Figure 8: Memory usage of operation and request objects

these implementations in a single code base and evaluated them for performance and memory usage. We observed that several scalability limitations exist in MPI implementations that prevent MPI from effectively supporting irregular applications at scale. Efficient solutions are needed to address these challenges.

References

[1] MPI Forum, "MPI: A Message Passing Interface Standard," 2015, http://www.mpi-forum.org/docs/docs.html.

[2] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. V. Dam, D. Wang, J. Nieplocha, E. Apr, T. L. Windus, and W. A. deJong, "NWChem: A Comprehensive and Scalable Open-source Solution for Large Scale Molecular Simulations," *Computer Physics Communications*, 2010.

[3] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, and J. Willcock, "Graph500," http://www.graph500.org/.

[4] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote Memory Access Programming in MPI-3," *TOPC'15*.

[5] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2004.

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

[7] OpenFabrics Alliance (OFA), " OpenFabrics Enterprise Distribution (OFED)," http://www.openfabrics.org/.

[8] The InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1, Release 1.2," 2004.

[9] Sandia National Laboratories, "Portals Network Programming Interface," http://www.cs.sandia.gov/Portals/.

[10] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. Atos, "The BXI Interconnect Architecture," in *HOTI'15*.

[11] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D Mesh/-Torus Interconnect for Exascale Computers," *IEEE Computer*.

[12] 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 *SC'12*.

[13] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *SC'11*.

[14] Mellanox Technologies, "RDMA over Converged Ethernet (RoCE) - An Efficient, Low-cost, Zero Copy Implementation," http://www.mellanox.com/page/products_dyn?product_family=79.

[15] Fujitsu Limited, "FUJITSU Supercomputer PRIMEHPC FX100 Evolution to the Next Generation," https://www.fujitsu.com/global/Images/primehpc-fx100-hard-en.pdf.

[16] Argonne National Lab, "MPICH," https://www.mpich.org/.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI implementation," in *Euro PVM/MPI'04*.

[18] J. Liu, J. Wu, S. Kini, P. Wyckoff, D. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," in *ICS'03*.

[19] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided," in *SC'13*.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *ISCA'92*.

[21] R. Thakur, W. Gropp, and B. Toonen, "Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication," *IJHPCA'05*.