Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication

Joseph Schuchart
schuchart@icl.utk.edu
Innovative Computing Laboratory (ICL), University of Tennessee Knoxville (UTK)
Knoxville, TN, U.S.A.

José Gracia
gracia@hlrs.de
High-Performance Computing Center Stuttgart (HLRS)
Stuttgart, Germany

Christoph Niethammer
niethammer@hlrs.de
High-Performance Computing Center Stuttgart (HLRS)
Stuttgart, Germany

George Bosilca
bosilca@icl.utk.edu
Innovative Computing Laboratory (ICL), University of Tennessee Knoxville (UTK)
Knoxville, TN, U.S.A.

1 INTRODUCTION

Modern high-performance networks commonly provide the capability to directly access memory on a remote host for reading, writing, and atomic memory updates [2, 23]. The hardware is capable of transferring data without the involvement of the CPU at the target node once the upper software layers have properly set up the parameters for the transfer, e.g., registered memory with the network interface card (NIC) and exchanged the registration information with the peers involved. MPI implementations typically make use of these low-level network features to provide efficient transfer of large messages between peers communicating through point-to-point or collective operations [36].

The MPI RMA interface was introduced with MPI version 2.0 [24] and has seen a major overhaul in version 3.0 [25], including the addition of allocated and dynamic windows. The intention of this interface is to expose the network’s low-level remote direct memory access (RDMA) capabilities to the user by providing procedures for put, get, and accumulate operations on windows that encapsulate memory for which registration information has been exchanged among the group of participating peers. By using MPI RMA, applications are able to decouple communication and synchronization, e.g., to perform bursts of communication before synchronizing through collective and point-to-point communication or by setting a signal flag at the target using accumulate operations.

In its current form, an MPI window is an object spanning across the processes in its group, i.e., its creation and destruction are collective operations. Active target synchronization involves collective operations. Passive target synchronization, on the other hand, only involves specific MPI calls at the origin of the operation (some MPI implementation may, however, depend on the target to call into MPI procedure to progress outstanding RMA operation). With the exception of dynamic windows, the memory accessed through these windows is static, requiring collective (re)allocations to increase the amount of memory accessible to RMA operations. Dynamic windows, on the other hand, allow for dynamically attaching and detaching memory segments, albeit at a significant penalty in performance, which will be discussed in Section 4.1.

Interest in RMA in the user community seems to be growing [4, 8]. However, the RMA chapter has seen little change during the work on the 4.0 release of the standard, despite there being several
known shortcomings that inhibit full and efficient usage of MPI RMA in applications and runtime systems [30, 32]. In this paper, we draw from our experience in using MPI-3 RMA in the context of the DASH project [15, 32] in general and the global task synchronization scheme built on top of it in particular [31]. We will discuss a number of issues found during this work and propose potential remedies that mostly consist in allowing the user to express the anticipated use of the RMA interface to the implementation through additional info keys (Section 2). In order to increase flexibility in applying these configurations, we propose an extension to the MPI RMA interface that allows users to duplicate windows, accessing the same memory using the same network resources but with configurations adjusted to the needs of different regions of the application (Section 3). In order to increase flexibility in the use of RDMA through the MPI RMA interface, we propose the addition of MPI memory handles that allow users to explicitly manage the registration information of memory attached to dynamic windows, alleviating the performance penalties that stem from the current design of dynamic windows (Section 4).

A brief discussion of additional improvements and changes that are beyond the scope of this work will be provided in Section 5. Section 6 discusses some of the implementation details for the proposed solutions before an evaluation of some of the proposals using micro benchmarks is presented in Section 7. Related work is discussed in Section 8 and conclusions are drawn in Section 9.

2 ADDITIONAL USER-PROVIDED INFORMATION

2.1 Thread-Scope Synchronization

Communication and synchronization in MPI RMA happens at the scope of processes, which encapsulate the memory made accessible to remote peers. Thus, a flush in passive target synchronization or a fence in active target synchronization ensure the completion of all operations previously issued by any thread in the current process to complete in the target process memory. Active target synchronization is collective either over the group of a window (MPI_Win_fence) or an otherwise provided group (post-start-complete-wait, PSCW). With passive target synchronization, flushes are local operations. Despite previous attempts at thread-specific communication endpoints [11], collective operations in MPI happen at the scope of processes. We will thus focus on the passive target synchronization.

In multi-threaded applications, individual threads may perform RMA operations independently. However, a thread calling into MPI_Win_flush potentially waits for the completion of operations issued by all threads of the same process to the same target, even though the completion of operations previously issued by the current thread might be sufficient for the application. This is depicted in Figure 1a, where the flush of Thread 1 is prolonged by operations issued by Thread 2. MPI implementations may use thread-specific network hardware resources (endpoints or raids) to reduce synchronization between threads when issuing RMA operations, e.g., by using one endpoint per thread or distributing operations across a fixed number of endpoints [17].

MPI supports flushing operations to a single target or to all targets in the group of the window, either with local or remote completion. We will focus on operations with remote completion, although the proposal also applies to flushes with local completion.

In order to restrict the completion semantics of flush operations to operations previously issued by the calling thread, the user has to signal this intention to the implementation. While it would be feasible to introduce a new set of functions to accomplish this goal, the required extension of the API would force implementations to provide such functionality even if support for multi-threaded RMA was limited. It would also introduce the notion of thread-scope operations into an API that is otherwise oblivious of the existence of multiple threads of execution, with the exception of MPI_Init_thread used to signal their (anticipated) existence.

We thus propose the addition of an info key called mpi_win_scope, which specifies the scope of synchronization operations on a window. If the value of that key is set to process (the default), flushes behave as today with operations issued by all threads required to complete during a flush. However, if the scope is set to thread, the implementation is free to restrict the scope of a flush to the operations previously issued by the calling thread, as depicted in Figure 1b. Since the process scope is a superset of the thread scope, implementations ignoring this info key remain correct. Users can check the support for the thread scope by querying the value associated with that key from the info object attached to the window usingMPI_Win_get_info [26, §12.2.7].

With the mpi_win_scope key set to thread on a window, implementations using thread-local endpoints only need to wait for the completion of operations on the endpoint assigned to the calling thread, potentially avoiding any synchronization between threads using RMA with passive target synchronization. The key has no effect on active target synchronization, since collective operations always happen at the process scope.

2.2 Operation Ordering

By default, MPI RMA guarantees the ordering of consecutive accumulate operations on the same memory location with the same data type and allows users to relax this constraint using the accumulate_ordering info key. However, in order to achieve ordering of put and get operations or to order accumulate operations to distinct memory location within a window, the application is required...
We have previously proposed adding a function waiting for completion of all operations if thread-specific network which would translate either into a memory barrier or a fence in the network interface card. In multi-threaded applications, however, the thread-local scope described previously ordering can be constrained to operations issued by the same thread. Applications relying on ordering using this info key are required to check whether the operation ordering using the mpi_win_order info key is supported and fall back to flushes in between operations otherwise. Listing 2 provides a modified version of the example in Listing 1 with the mpi_win_order set to true, avoiding the first flush by chaining two operations.

2.3 Hardware Accumulate Operations

MPI RMA accumulate operations are notoriously hard to implement efficiently: on the one hand, single-element operations such as MPI_Fetch_and_op and MPI_Compare_and_swap may benefit from the use of hardware atomic operations provided by the NIC due to the low latency of single-element operations implemented by the hardware. On the other hand, MPI_Accumulate allows users to issue operations on an arbitrary number of elements at the same time, which could potentially benefit from the higher bandwidth of operations performed by vector units on the host CPU [37]. The MPI standard requires implementations to provide element-wise atomicity of operations applied to the same memory location if using the same data type, regardless of whether they were issued through single-element operations or as part of a multi-element operation. In addition, not all possible operations are supported by the network hardware. For example, while networks commonly support addition and subtraction of integral values, support for integral value multiplication or operations on floating point values is often missing. Taken together, implementations once again cannot anticipate the operations that will be issued by the user and thus have to leave certain hardware features lay bare.

Implementation typically use two possible approaches: i) taking a lock at the target process before fetching the data, applying the operation, and writing the data back before releasing the lock; and ii) using active messages to transfer the data to the target and relying on the target CPU to perform the operation. Both approaches require the serialization of concurrent accumulate operations through some form of mutual exclusion device.

Listing 1: Using an atomic increment to signal the completion of a put, overlapping communication with useful work.

```c
int flag, one = 1;
MPI_Request req;
MPI_Rput(..., target, win, &req);
do {
    do_useful_work();
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
} while (!flag);
/* Flush needed for remote completion */
MPI_Win_flush(target, win);
/* Signal that the data has been written by incrementing a counter at the target */
MPI_Raccumulate(&one, ..., target, ..., win, &req);
do {
    do_useful_work();
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
} while (!flag);
```

Listing 2: The example of Listing 1 with mpi_win_order set to true, avoiding the first flush by chaining two operations.

```c
int flag, one = 1;
MPI_Request req;
MPI_Put(..., target, win);
/* Signal that the data has been written by incrementing a counter at the target */
MPI_Raccumulate(&one, ..., target, ..., win, &req);
do {
    do_useful_work();
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
} while (!flag);
```
Listing 3: Function to query the use of intrinsic hardware operations for a given set of operations on a number of elements of a certain data type.

An existing proposal to tackle this problem allows the user to specify how many elements will be used with which operations [35]. However, users would still have no information on whether accumulate operations will be executed in hardware or software. Some applications may rely on low-latency accumulate operations [6, 30] but the lack of transparency prevents them from picking an alternative algorithm if available. The proposed unidirectional signaling is hence not sufficient.

We propose the addition of a procedure (based on a previous suggestion in the MPI RMA working group [10]) that i) allows the application to query the implementation’s approach to performing a given accumulate operations for a given number of elements of a certain data type; and ii) to signal the anticipated usage pattern to the implementation. We borrow a concept from the C++ std::atomic wrapper type [20], which allows developers to query whether atomic modification of a given wrapped type is lock-free using the compile-time is_lock_free trait that signals whether a mutex or CPU-provided atomic operations are used to ensure atomicity. Contrary to std::atomic, MPI accumulate operations may apply to multiple elements at once and implementations may use a threshold for switching between hardware and software approaches. Thus, the information has to be query-able at runtime, for which we propose a new procedure called MPI_Win_op_intrinsic listed in Listing 3. For a given tuple describing the set of anticipated operations to be performed (ops) on the provided number elements (max_count) of a certain type on a specific window win, the implementation returns whether the operations will be performed with hardware operations intrinsic to the origin node, i.e., without relying on the participation of a CPU at the target. The set of operations are described as a string containing a comma-delimited list of operations, using the second half of the name of predefined MPI_Op elements (e.g., "sum"), "replace" for MPI_REPLACE, and "cas" to denote MPI_Compare_and_swap.

The information obtained from a call to MPI_Win_op_intrinsic may then be used to set a new boolean info key called mpi_assert_accumulate_intrinsic. If set to true, the application asserts that it will only issue accumulate operations in configurations for which the implementations has signaled the use of intrinsic operations. The results of the application disregarding this assertion are undefined, leading to modifications that are potentially non-atomic.

3 DUPLICATING MPI WINDOWS

In the previous section we have proposed three new info keys to help the user express the intended use of RMA operations on a window: restricting the scope of flushes to operations issued by the calling thread; ordering the operations issued on the window; and limiting the number of elements in accumulate operations to allow for the use of operations intrinsic to the origin hardware. However, changing the value of an info key overwrites the old value and thus makes it impossible to use different configurations concurrently, e.g., to request operation ordering on some threads but not on others or to use bandwidth-optimized accumulate operations on one part of the memory while using latency-optimized accumulate operations on another part of the window. Switching between these settings would require careful orchestration of info key values.

While it is legitimate to allocate memory in MPI_Win_allocate and pass that memory into a call to MPI_Win_create with different info key values, the resulting two windows are semantically independent with independent passive and active target synchronization semantics and no cross-window atomicity guarantees, as depicted in Figure 2a. In order to ease the task of managing different means of access to the same window memory and to keep windows with different info values in sync, we propose to add a window duplication function, as outlined in Listing 4. In contrast to two independently created windows, duplicated windows may share internal data structures and window memory while carrying potentially different access semantics. Duplicated windows can thus be considered as different handles to the same underlying memory and network resources, as depicted in Figure 2b.

This approach enables the use case described above in which different window settings are used in different parts of an application, while sharing the underlying resources. Some restrictions apply: as long as the duplicated windows use the same value for mpi_assert_accumulate_intrinsic accumulate operations are...
atomic across these windows. However, issuing accumulate operations on two windows having different values for the mpi_assert_accumulate_intrinsic is legal but the accumulate operations may not be atomic with respect to each other. It is up to the user to coordinate the correct use of this info key.

We propose a new function called MPIX_Win_dup_with_info that is used to duplicate the window with a new set of info keys. Its signature is shown in Listing 4. Info keys from the parent window will be duplicated into the new window, with the provided info keys overriding existing ones.

Since the original and duplicated windows are not logically separate, all synchronization operations applied to a window also apply to its duplicates, and vice versa. For example, the duplicated window may not be locked if the parent window has already been locked. In essence, window duplication is akin to assignment of an MPI_Win variable with the added ability to control certain info keys.

The call to MPIX_Win_dup_with_info is a local operation and thus does not entail any synchronization with other processes in the group of the parent window. An MPI implementation may not be able to change certain info keys during this call and may thus reject the change by retaining the original or default value. Users should check whether the MPI implementation is able to support the requested configuration by querying the active info keys using MPI_Win_get_info.

4 DYNAMIC MEMORY HANDLES

In its current form, window creation is a collective operation in MPI. With the exception of dynamic windows, MPI windows and their memory are statically allocated or assigned, which allows the MPI implementation to exchange all relevant connection and registration information during window creation and enables the use of the network’s RDMA capabilities (Figure 3a). However, such static windows may be impractical if an application’s communication requirements changes over time, requiring repeated (collective) reallocation of windows. Moreover, applications may treat communication and memory allocation as orthogonal concerns such that the use of window memory would break through abstraction boundaries, requiring major restructuring efforts to integrate the allocation of memory through MPI windows.

In contrast, dynamic windows, as their name suggests, allow users to attach and detach memory dynamically in a local operation after the window has been created in a collective operation. Attaching the memory explicitly allows the MPI implementation to register the memory with the network device for later access using RDMA. Addressing in dynamic windows is done using absolute virtual addresses: after attaching memory to the dynamic window, the process distributes the virtual address that is then used as displacement at the origin of an RMA operation.

Unfortunately, the use of virtual addresses is the greatest weakness of dynamic windows: in contrast to static windows, the origin of an RMA operation initially has no information on the underlying memory registration and thus has to either query this information from the target before issuing RDMA operations (Figure 3b) or fall-back to emulating remote memory accesses using active messages (AM, Figure 3c). Both approaches add considerable latency, especially to RMA operations on small amounts of data. Since the target may detach and reattach the same virtual base address, the registration information for the same virtual address may change at the target in between RMA operations at the origin. Thus, while caching techniques are possible, the origin has to at least verify the validity of the cached registration information on every RMA communication operation. The lack of life-time guarantees is thus the main reason for added overhead when using dynamic windows.

We will show in Section 4.1 that the difference in latency between allocated and dynamic windows is significant on all tested MPI implementations. We will then propose an extension to the MPI RMA interface to allow applications to explicitly exchange registration information and thus make life-time guarantees to the MPI implementation that enable it to use RDMA with zero overhead.

4.1 State of the Art

Table 1 lists the software used for comparison of dynamic and allocated windows in this section. All measurements were conducted on an HPE Apollo 6500 system Hawk installed at HLRS. The nodes are equipped with dual-socket 64-core AMD EPYC 7742 processors and connected through Mellanox InfiniBand HDR200 in a 9D hyper-cube fabric.

4.1.1 Communication Latency. Figure 4 shows the latency of put operations measured using the OSU benchmark osu_put latency benchmark on different MPI implementations. While the latencies on allocated windows are similar across the three implementations, the differences are significant for dynamic windows. Especially for smaller transfer sizes, the penalty of using dynamic windows over allocated windows ranges from a factor of 1.5× (MPICH, MVAPICH) over 3× (Open MPI using UCX). As described earlier, this discrepancy between allocated and dynamic windows stems from the missing registration information, which either leads to a fall-back to AM-based emulation or requires fetching the registration information before issuing the actual operation.
Table 1: Software configuration.

| Software   | Version  | Configuration/Remarks |
|------------|----------|-----------------------|
| Open MPI   | 4.0.5    | –with-ucx=...         |
| MPICH      | v4.0.0.a | –with-device=ch4:ucx  |
| MVAPICH    | 2.3.5    | –with-device=ch3:mail |
|            |          | –with-rdma=gen2       |
| UCX        | 1.10.0   | –enable=mt            |
| GCC        | 10.2.0   | site installation     |
| OSU Benchmarks | 5.6.2  | none                  |

Figure 4: Latency of put operations using allocated and dynamic windows.

4.1.2 One-Sided Progress Behavior. In order to better understand the behavior of the implementations, we repeat a benchmark here that was used in [30] to determine the one-sided behavior of various accumulate MPI implementations. In this test, the target process is busy outside of MPI for a fixed amount of time before waiting in an MPI barrier for the origin to complete the execution of a number of RMA operations. For the results shown in Figure 5, the origin performs \( n = 100,000 \) put operations, each followed by a flush, while the target is busy outside of MPI for \( t = 3 \) s. Thus, a latency of \( \frac{n}{t} > 30 \mu s \) indicates that the origin is not progressing until the target enters the MPI barrier. As can be seen in Figure 5, both MPICH and MVAPICH lack progress for dynamic windows, indicating an implementation relying on the target CPU to execute active messages (Figure 3c). Operations on dynamic windows using the UCX integration in Open MPI, on the other hand, progress, albeit at a significantly higher latency, as discussed in the previous section. We note that while AM-based emulation may yield sufficiently low-latencies in benchmarks such as the osu_get_latency, in practice it renders the performance of MPI RMA unpredictable as performance depends on the behavior of the target process, (partially) defeating the purpose of a one-sided programming interface.

4.2 Life-Time Control Through Memory Handles

Allowing users to provide MPI directly with registration information on memory to be accessed through windows provides both life-time information and avoid additional overhead during communication operations. We propose the following three additions to the MPI RMA interface (their signatures are detailed in Listing 5):

- **MPIX_Memhandle_create** registers a memory region starting at base of size size with the provided dynamic window for later access through RMA operations. The function returns in memhandle a memory handle of size memhandle_size. The memhandle should be a buffer of at least MPI_MAX_MEMHANDLE_SIZE bytes. The memory handle contained in this buffer can be distributed to peer processes.

- **MPIX_Memhandle_release** Once all RMA operations have completed and the registered memory is not needed anymore (i.e., all peers have signaled completion) the memory handle can be released using this function. A call to this function, no more RMA operations may be issued on windows created from this memory handle. The corresponding windows must be freed through a call to MPI_Win_free.

Instead of sending the virtual address of an attached memory region to the peer the application now sends the registration information directly, which is an opaque data structure that is specific to the underlying implementation (which may differ from platform to platform for the same MPI implementation).

The call to **MPIX_Memhandle_release** is a local operation and the resulting window remains connected to its parent window. We restrict synchronization of such windows to passive target synchronization and require the lock and unlock to be applied on the parent dynamic window. These restrictions allow the MPI implementation to avoid allocating additional internal memory during the creation of the memory handle required to handle these synchronization operations. Thus, the only operations permitted on memory handle windows are put, get, and accumulate operations as well as flushes. We expect users to use shared locks and rely on other synchronization and signaling mechanisms such as collective operations, point-to-point operations, or accumulate operations. By allocating a separate window object, the implementation is...

Figure 5: Average latency of 100,000 single-byte put and flush with the target sleeping for 3 s. Latencies above 30 \( \mu s \) indicate no progress while the target process is not executing MPI calls.
not required to maintain and repeatedly traverse a list of attached memory handles but instead the resulting window identifies the remote memory region directly and allows for the implementation to stop tracking that remote memory region once the applications calls MPI_Win_free on the memory handle window.

4.3 Example

An example for how we envision memory handles to be used is provided in Figure 6. After creating the window win from the memory handle received from Process A, Process B performs an arbitrary number of RMA operations on the corresponding target memory region before signaling completion back to Process A, which then releases the memory handle. We will show in Section 7.3 that latencies using memory handle windows are on par with allocated windows in our proof-of-concept implementation.

5 FURTHER IMPROVEMENTS

In this section we briefly discuss efforts that we consider beneficial for the future direction of the MPI RMA API but that are beyond the scope of this paper.

5.1 Completion Notification

Past work has focused on completion notification at the target and we support the approach presented in [34], which was based on a previous proposal [3]. However, we caution that instead of introducing new test/wait routines (MPIX_Win_test_notify and MPIX_Win_wait_notify) the notification should integrate with the existing request facilities. Due to the nature of progress in MPI, these functions have to ensure progress in order to drive non-RMA communication operations on whose completion at the origin the notification may depend, e.g., collectives or point-to-point operations.

Using a request-based notification mechanism (without relying on persistent requests [3]) allows users to test or wait for completion of all MPI-related communication, including RMA notifications. We envision a function such as MPI_Win_wait_notify (win, notify_id, request) that returns a request, which can later be used with the regular request test and wait infrastructure in MPI. We leave an in-depth investigation into such an interface for future work.

5.2 Remote-Completing Request-Based Operations

MPI RMA provides variants of put, get, accumulate, and get-accumulate that return a request that can be used to test and wait for completion of that particular operation. However, the completion of a request returned by MPI_Rput, MPI_Raccumulate, and MPI_Rget_accumulate only signals local completion, requiring a subsequent flush to achieve remote completion before signaling completion to the target. This flush, in turn, may wait for the completion of unrelated operations, potentially from other threads. In many cases, MPI implementations are able to implement request-based put with remote completion more efficiently than the application using a flush (e.g., by leveraging guarantees of the underlying transport library). We thus propose adding functions MPI_Rrput (remote-completing request-base put), MPI_Rraccumulate, and MPI_Rrget_accumulate with similar signatures as their current request-based counterparts. The completion of requests provided by these procedures signal both local and remote completion, alleviating the need for an additional flush and thus potentially improving the efficiency of applications using request-based operations. An example of the case described above using a remote-completing request-based put is given in Listing 6. If a regular call to MPI_Rput was used instead, a flush would be necessary in Line 8 before the call to MPI_Allreduce was used to signal completion to all peers.

5.3 Deprecating Active Target Synchronization

We encourage efforts to engage with users of the active target synchronization interface to identify potential road-blocks in the transition to passive target synchronization, with the goal of phasing
Listing 6: The example of Listing 1 using a remote-completing request-based put and a collective operation for synchronization.

```c
int flag, one = 1;
MPI_Request req;
MPI_Rput(..., target, win, &req);
do {
  do_useful_work();
  MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
do
  if (flag) {
    MPI_Allreduce(..., another_variable, ...);
  }
}
```

Figure 7: Flushes in the UCX one-sided communication module in Open MPI. Threads calling MPI_Win_flush_all iterate over the endpoints of all threads to ensure completion of all operations issued by the process.

out active target synchronization. By eventually removing active target synchronization, the RMA part of the standard would become cleaner and more concise, removing a significant portion of its complexity and providing easier access to the one-sided communication chapter. Moreover, we believe that the collective nature of active synchronization incurs excessive overhead and techniques mentioned earlier in this section may achieve similar goals more efficiently. However, such an effort has to be undertaken in collaboration with the community, which is beyond the scope of this paper. Previous surveys of MPI usage might help in identifying the relevant user groups [4, 21].

6 IMPLEMENTATION

6.1 Reference Implementation

The used reference implementation (the UCX one-sided communication module in Open MPI’s main development branch) employs thread-specific UCX worker objects for each window. Since UCX endpoints are specific to a worker, each thread also manages its own set of endpoints (connections with peers in the window) that are created upon the first access to that peer in the window and used to issue RMA operations. The worker and endpoints are stored in lists in the window, over which threads iterate during flushes on that window, as depicted in Figure 7. Access to that list and to each thread’s connection information are protected through mutexes to ensure thread-safety. While this scheme allows threads to issue operations on independent endpoints, it leads to significant synchronization overheads during a flush operation.

Our proof-of-concept implementations are based on this infrastructure and we note that other implementations may have different approaches, leading to different degrees of effectiveness of the proposed RMA extensions.

6.2 Thread-Scope Flushes

The aforementioned reference implementation allowed for an easy implementation of thread-scope flushes discussed in Section 2.1: instead of iterating over the list of workers or endpoints, a thread calling into a flush only operates on its local worker or endpoint. Thus, no access to workers or endpoints owned by other threads is necessary, greatly reducing the amount of both the amount of work and inter-thread synchronization required to achieve completion of operations issued by an individual thread.

6.3 Operation Ordering

When issuing RMA operations on windows for which the mpi_win_order info key discussed in Section 2.2 is set to true, the calling thread calls into ucp_worker_fence on the used UCX worker before issuing the actual operation. This function call ensures that the operation will complete at the target only after all previously issued operations have completed at the target.

The used UCX worker depends on the scope set for the window. Since ucp_worker_fence guarantees operation ordering for a specific worker only, all operations are funneled through the endpoints of a single worker when the process scope is enabled on the window (the default). While this may incur additional synchronization between threads, it allows for operation ordering without explicitly waiting for operations to complete using a full flush. With thread scope enabled, the calling thread invokes `ucp_worker_fence` on its local worker.

6.4 Hardware Accumulate Operations

Open MPI already provides support for an info key on windows, called acc_single_intrinsic, that allows users to signal to the implementation that only single-element accumulate operations with support for intrinsic operations will be used. Its effectiveness for ensuring low-latency accumulate operation on supported hardware has been shown in [30]. Due to space limitations, we refrain from any further discussion of both the implementation and its effectiveness as the proposal presented in Section 2.3 provides the same guarantees as the existing acc_single_intrinsic info key.

6.5 Memory Handle Windows

In contrast to collectively allocated windows, memory handle windows only provide access to a single memory region at a specific target process. It is thus sufficient to store the information for that target in the window. Our implementation employs the parent window UCX worker and endpoint information described in Section 6.1 for communication and only stores the memory handle’s

---

3 The proof-of-concept implementation for the thread-scope and operation ordering info keys can be found at https://github.com/devreal/ompi/tree/mpi-win-dup-with-info.

4 The proof-of-concept implementation for memory handle windows can be found at https://github.com/devreal/ompi/tree/one-win-memhandle-parentwin.
7 EVALUATION

7.1 Thread-Scope Flushes

We use the RMA-MT benchmark to measure latencies of RMA operations in a multi-threaded context [13]. The existing benchmark covers both active and passive target synchronization, with multiple worker threads issuing RMA operations and the main thread performing the ensuing RMA synchronization. While this pattern may be useful for fork-join thread models such as OpenMP work-sharing loops, it is inadequate for task-based applications that typically do not exhibit synchronization points. We have thus extended the benchmark to include a variant in which threads perform RMA operations followed by flushes, a pattern that is commonly found in applications using puts and flushes to ensure remote completion.

The latencies of a put followed by a flush when selecting either thread- or process-scope using the mpi_win_order info key (Section 2.1) with Open MPI as well as using MPICH and MVAPICH are shown in Figure 8. The slightly higher latency of process-scope flushes in the case of a single worker thread shown in Figure 8a for Open MPI can be explained by the fact that the single worker thread has to perform a flush on its endpoint and the main thread’s endpoint, which is not required when using thread-scope flushes. Overall, however, the latencies of the different implementations are mostly similar. By contrast, for 32 worker threads shown in Figure 8b, the use of thread-scope flushes leads to an order of magnitude lower latencies for small transfer sizes compared to MPICH and MVAPICH and close to two orders of magnitude for Open MPI. For larger transfer sizes a factor of two is achieved.

The latencies of a put followed by a flush when selecting either thread- or process-scope using the mpi_win_order info key (Section 2.1) with Open MPI as well as using MPICH and MVAPICH are shown in Figure 8. The slightly higher latency of process-scope flushes in the case of a single worker thread shown in Figure 8a for Open MPI can be explained by the fact that the single worker thread has to perform a flush on its endpoint and the main thread’s endpoint, which is not required when using thread-scope flushes. Overall, however, the latencies of the different implementations are mostly similar. By contrast, for 32 worker threads shown in Figure 8b, the use of thread-scope flushes leads to an order of magnitude lower latencies for small transfer sizes compared to MPICH and MVAPICH and close to two orders of magnitude for Open MPI. For larger transfer sizes a factor of two is achieved.

The thread-scaling behavior for single-byte transfer sizes is shown in Figure 9. The benefit of using thread-scope flushes (where appropriate) becomes clear, as it reduces the amount of work each thread has to perform inside a flush, reducing the inter-thread synchronization to a minimum.

7.2 Operation Ordering

We have modified the osu_put_latency benchmark to include an option suppressing intermediate flushes, i.e., puts are issued in a loop and synchronization happens during MPI_Win_unlock after the specified number of operations have been started. This allows us to better observe the overhead of operation ordering. Figure 10 shows the latency for regular put and flush as well as the variant without intermediate synchronization, both with and without operation ordering enabled using the mpi_win_order info key discussed in

![Figure 8: Latency of multi-threaded put and flush with process-scope and thread-scope flushes.](image1)

![Figure 9: Latency of single-byte put and flush with process-scope and thread-scope when scaling the number of threads.](image2)

![Figure 10: Latency of put with flush as well as put without intermediate synchronization but ordering enabled using the mpi_win_order info key.](image3)

![Figure 11: Latencies of put and flush using 32 worker threads with and without operation ordering enabled.](image4)
Section 2.2. While some additional latency can be observed due to the requested ordering, the latency of ordered puts is still significantly lower than the if flushes were used to enforce ordering of RMA operations. While this is by no means surprising (flushes likely incur at least the latency of a full network round trip), it underscores that the MPI RMA interface should provide means for ordering operations beyond waiting for completion.

We use the same DMA-MT benchmark to compare the impact of operation ordering using the mpi_win_order info key discussed in Section 2.2 on the latency of put operations. With 32 worker threads (Figure 11), enabling operation ordering with process-scope flushes reduces latencies since, as described in Section 6.3, all operations are funneled through a single endpoint.

7.3 Memory Handles

Similar to the previous results, the latency of puts measured using the osu_put_latency using allocated, dynamic, and memory handle windows are shown in Figure 12. The difference between allocated and windows created from memory handles is negligible. Compared to the latencies of today’s dynamic windows discussed in Section 4, the benefits of combining the flexibility of dynamic windows with the use of direct RDMA without the additional overhead of querying registration information become clear.

Latencies with added window creation and destruction are included in Figure 12, adding approximately 1 μs and still being significantly lower than the latencies for existing dynamic windows while employing the network’s RDMA capabilities. We believe that such an overhead is sufficiently low to allow applications to rapidly create and destroy memory handle windows for use with RMA operations.

8 RELATED WORK

Several improvements to MPI’s ability of handling multi-threaded communication has been proposed over the years, ranging from thread-safe probes [18] over thread-specific endpoints [11] to partitioned communication [16] and (most recently) the use of continuations [29, 33]. Work on improved implementation support for multi-threaded MPI in general [28] and RMA in particular [17] has also been described. The thread-scope flushes proposed in this work provide additional information to the implementation to better leverage some of that earlier work.

Several abstractions for one-sided communication provide collective allocation capabilities, including OpenSHMEM [7, 27] and GASNet [5], but lack local allocation of exposed memory. Lower-level PGAS abstractions such as GASPI [1] and LCI [9] provide dynamic local allocation of exposed memory. The memory handle windows proposed in this work aim at closing the gap to these low-level abstractions and increase the flexibility of MPI RMA.

OpenSHMEM has introduced so-called contexts to provide isolation between threads, at the cost of significant extension of the API [12]. The proposed duplicate windows with thread-scope setting is an attempt to achieve a similar to goal, without significantly extending the RMA interface.

The proposed memory handle windows may be useful for applications to work around limitations of hardware tag matching engines [22] by reducing the number of exchanged messages, e.g., by organizing multiple data transfers through MPI RMA and using matched messages for signaling purposes only. The proposed info key for ordering RMA operations might enable implementations to utilize triggered operations on network hardware, which already have proven useful in the implementation of collective operations [19] and in the implementation of fence operations in OpenSHMEM [14].

9 CONCLUSIONS

We have identified several short-comings of the RMA part of the current MPI standard versions that potentially cause low performance due to high costs of synchronization and a lack of usage of available hardware resources in RMA operations. By allowing users to provide additional information on the anticipated usage of windows, implementations can adapt to the application’s usage patterns, enabling improved performance, e.g., by constraining the scope of flushes, reducing the number of flushes by enforcing the ordering of operations, and by constraining the number of elements in accumulate operations. By introducing the duplication of windows, we provide a way for users to maintain differently configured handles to the same window resources, facilitating easy switching between configurations in different parts of an application. Additionally, we propose to add the notion of memory handles to the MPI RMA interface, enabling bare-metal performance of dynamic windows by allowing users to manage memory registration information and provide life-time guarantees of memory segments, which eliminates costly querying at the target before performing RMA operations. Our benchmarks show that the proposed additions to the RMA chapter can greatly reduce the synchronization overhead, allowing applications to make better use of the hardware capabilities through the MPI RMA interface.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union’s Horizon 2020 research and innovation programme under the ChEESE project, grant agreement No. 823844. This material is based upon work supported by the National Science Foundation under Grant No. #1664142 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration.
