MPI-Interoperable Generalized Active Messages

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

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

***Abstract*—Data-intensive applications have become increasingly important in recent years, yet traditional data movement approaches for scientific computation are not well suited for such applications. The Active Message (AM) model is an alternative communication paradigm that is better suited for such applications by allowing computation to be dynamically moved closer to data. Given the wide usage of MPI in scientific computing, enabling an MPI-interoperable AM paradigm would allow traditional applications to incrementally start utilizing AMs in portions of their applications, thus eliminating the programming effort of rewriting entire applications. In our previous work, we extended the MPI_ACCUMULATE and MPI_GET_ACCUMULATE operations in the MPI standard to support AMs. However, the semantics of accumulate-style AMs are fundamentally restricted by the semantics of MPI_ACCUMULATE and MPI_GET_ACCUMULATE, which were not designed to support the AM model. In this paper, we present a new generalized framework for MPI-interoperable AMs that can alleviate those restrictions, thus providing a richer semantics to accommodate a wide variety of application computational patterns. Together with a new API, we present a detailed description of the correctness semantics of this functionality and a reference implementation that demonstrates how various API choices affect the flexibility provided to the MPI implementation and consequently its performance.**

I. Introduction

In recent years, many new, data-intensive applications have become prominent in various domains such as bioinformatics and social network analysis. A fundamental characteristic of these applications is that they process a large amount of data, possibly in irregular patterns, requiring computation and data movement to be carefully balanced for achieving high performance. Traditional programming approaches that were designed for environments where computation is regular and its cost is typically significantly larger than the data movement cost are not well suited for such applications. Alternative programming frameworks are desirable.

The Active Messages (AM) model [1] is a parallel programming paradigm that can potentially bridge this gap. It allows a process to specify a function handler to be executed when a particular type of message is received, thus relieving itself of the responsibility to explicitly receive and process the message. Such a model can be more natural to use in some, though not necessarily all, scenarios. A combination of the traditional MPI SEND/RECV- or PUT/GET-like model to move data and an AM-based model to move computation, however, can provide applications and high-level libraries with the necessary tool set to efficiently and dynamically balance their computation and data movement workload in order to achieve high performance.

In [2], we proposed an asynchronous and MPI-interoperable framework for AMs by leveraging the MPI remote memory access (RMA) model. The framework extended the MPI-3 accumulate operations to support user-defined functions at the target. While such a framework demonstrates the potential of MPI-interoperable AMs, it is restricted in a number of ways, including its ability to describe complex data layouts on which the AM can compute, how data associated with the AM is transmitted and staged at the remote process, what AMs can execute concurrently, and what ordering and memory consistency guarantees are available.

In this paper, we propose a generalized framework for MPI-interoperable AMs. Specifically, we propose a new set of functionality and semantics that no longer rely on MPI-accumulate operations but still maintain complete compatibility with the MPI-3 standard. The new semantics allow the implementation to achieve better performance, for example by controlling streaming and data usage granularity, concurrency capabilities among AMs, and ordering semantics. In addition to the design description, the paper presents a reference implementation of the generalized framework and an evaluation demonstrating the performance impact of the various semantic choices.

Recommended reading: While this paper provides some background on the MPI RMA semantics, it is not meant to be a comprehensive description. We highly recommend that the reader of this paper also read through past papers and books that more thoroughly discuss these semantics (e.g., [3], [4], [2]) in order to better understand the subtle characteristics and capabilities of MPI RMA on which this paper relies.

Naming Convention: We prefix all functions that are defined in the MPI-3 standard with **MPI_**, while new functions that are proposed in this paper are prefixed with **MPIX_** (MPI extensions). Also, we refer to accumulate-style operations that already exist in MPI-3 as "MPI-accumulate" operations, and we refer to our previous work that extends these operations to enable AMs as "AM-accumulate" operations.

II. Background and Related Work

In this section, we discuss the AM paradigm and our previous work on MPI-interoperable AMs.

A. Active Messages Paradigm

The AM paradigm [1] was proposed by von Eicken et al. for Split-C in 1992. With AMs, the sender of a message specifies a message handler to be executed at the receiver upon arrival of the message. When the message is received, the corresponding handler is triggered to process the data in that message. Unlike traditional two-sided message passing, the application on the receiver side does not need to explicitly call a function in order to receive the message. Previous libraries that support AMs include GASNet [5], IBM DCMF [6], IBM LAPI [7], and IBM PAMI [8]. While popular in high-performance computing systems, these libraries either do not maintain runtime compatibility with MPI or are platform specific. Thus, existing MPI applications cannot utilize them without necessarily duplicating runtime resources such as internal buffers or asynchronous progress threads.

Some work has also been done on supporting AMs on top of the MPI library, thus allowing existing MPI applications

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contracts DE-AC02-06CH11357, DE-FG02-08ER25835, and DE-SC0004131.

to utilize the AM paradigm. Examples include AM++ [9] and AMMPI [10]. While portable, they are restricted in a number of ways, including lack of asynchronous progress, inability to marshall/demarshall datatypes, and absence of explicit semantics for ordering, concurrency of AMs, and memory consistency.

B. Accumulate-Style Active Messages

In [2], we proposed a framework for asynchronous and MPI-interoperable AMs by extending **MPI_ACCUMULATE** and **MPI_GET_ACCUMULATE** to support user-defined operations. MPI allows applications to create user-defined function handlers and operations that correspond to these handlers. As of MPI-3, however, user-defined operations may only be used in collective communication functions such as **MPI_REDUCE** and cannot be used in **MPI_ACCUMULATE** and **MPI_GET_ACCUMULATE**. In our previous work, we extended such user-defined operations to be used within MPI-accumulate operations, thus imitating AMs in MPI. In particular, we were able to achieve such functionality by adding one additional function to MPI, **MPIX_AM_WIN_OP_REGISTER**, that collectively registers the functions associated with the operation.

We also classified MPI AMs into three classes: (1) NO-ASYNC, where asynchronous processing is not supported; (2) THREAD-ASYNC, where asynchronous processing is provided by using a thread above the MPI library (for example, if the application creates a thread for incoming messages, such usage would fall into this class); and (3) INTEGRATED-ASYNC, where asynchronous processing is supported within the MPI implementation transparent to the user. Our framework belongs to the third class. We also designed various techniques that allow the AM thread to block for network communication, while allowing shared-memory-based AMs to rely on an "origin computes" model. In this model, the origin can view the target window data (over shared memory) and compute the AMs on it, thus allowing the network AM thread to sleep when there are no AMs. While this framework allowed us to demonstrate how an MPI-interoperable AM framework can be designed, it has a number of usage restrictions resulting in programming complexity and memory inefficiency, as we demonstrate in Section III.

III. Restrictions of Accumulate-Style AMs

Because MPI-accumulate operations were not originally intended for AMs, their semantics do not quite match what AMs need. Consequently, AM-accumulate operations, which are based on MPI-accumulate operations, have several shortcomings that we analyze in this section. Based on this analysis, we propose a new AM model in Section IV.

A. Data Access

One restriction in using AM-accumulate stems from the way it represent data layouts. Specifically, the user-defined function (which was originally intended for **MPI_REDUCE**-like operations in MPI) accepts a single data layout (datatype and count). Thus, both the input and output buffers must have exactly the same layout. While this requirement is suitable for reduction operations, where multiple input arrays are reduced into a target array, it is too restrictive for AMs. For example, consider an application where the data on the target process is an array of bins of containers representing value ranges. When the process receives an AM with an integer that falls into a given bin, the corresponding counter is incremented. In such a model, the AM input data contains a single integer, while the AM handler needs access to the entire array of bins to do the necessary processing. Such operations cannot be handled by AM-accumulate.

A similar restriction arises with respect to the data that is fetched back to the origin process. While **MPI_GET_ACCUMULATE** provides such capability, its semantics require it to return the original data at the target before the operation was performed. Thus, the AM cannot return any arbitrary user-specified data. One example where such capability is needed is DNA sequence assembly applications (e.g., SWAP [11] and Kiki [12]). These applications rely on storing large databases of string sequences on distributed memory. A process that needs to search for a string sequence in the database would send the query sequence to the target as an AM, which would then search through its part of the database and return the matches. Although the AM requires access to the entire data on the target process, it does not necessarily require all of this data to be returned to the origin. Only the matches need to be returned. To emulate such a function, SWAP currently uses **MPI_SEND/RECV** with threads, potentially wasting cores waiting for incoming requests. Kiki uses dedicated "server processes" that only process such messages but perform no application computation, again wasting some user-defined number of cores for these processes. Applications in other domains, such as MADNESS [13] (computational chemistry), are similarly restricted and emulate AM functionality using **MPI_SEND/RECV** with threads.

B. Message Segmentation and Temporary Buffers

The MPI standard has deliberately not required the MPI implementation to have any temporary buffers. That is, the application cannot assume that the MPI implementation will use internal buffers for communication and has to be correct when no such buffers are available. For example, although most MPI implementations use "eager buffers" (internal temporary buffers) for communicating small messages and application buffers through a "rendezvous" hand-shake for communicating large messages, the MPI standard does not expose such modes to the application. Thus MPI implementations can experiment with different buffering techniques to improve performance.

This philosophy is also apparent in MPI-accumulate operations that guarantee atomicity only at the granularity of predefined datatypes. Thus, an MPI implementation that has no internal buffers can segment operations into multiple smaller operations (potentially at the granularity of one predefined datatype per operation) and issue them separately. Since MPI-3 allows only predefined computations to be used with MPI-accumulate operations, the MPI implementation knows these operations a priori and can implement them appropriately in order to compute on segmented data.

For AM-accumulate, however, since the computation is now defined by the user application, the MPI implementation cannot know what an appropriate segmentation granularity would be. Segmenting data to the granularity of a predefined datatype might be too restrictive for the application. For example, in the DNA assembly example given in Section III-A, if the MPI implementation segments the input DNA string sequence to the granularity of individual characters, the user function cannot search for the entire string in the target database with this limited information. At the other extreme, if the MPI implementation had to send all the input data to the target

process, it would be required to buffer arbitrarily large input data internally.

C. Lack of Concurrency

In AM-accumulate, concurrent execution of AMs is not well defined. In an environment where multiple origin processes simultaneously trigger AMs on the same target process, or where the same origin process triggers multiple AMs on the same target process, can the MPI implementation simultaneously execute the different AMs? MPI-accumulate operations are atomic at the granularity of predefined datatypes. That is, if two such operations target the same memory location on the same target, the MPI implementation will ensure that these updates do not clobber each other. With user-defined operations, however, MPI can no longer keep track of such atomicity. A conservative implementation of AM-accumulate could serialize all AMs computing on overlapping memory locations, thus forcing no concurrency in execution. However, such an implementation would be highly penalizing for performance. For example, for AMs that only read the target data but do not update it, no such atomicity is required, and the lack of concurrency can hurt performance.

D. Interoperation with Other MPI Messages

MPI-accumulate operations have well-defined interoperation semantics with other MPI messages. Multiple MPI-accumulate messages updating the same memory region are guaranteed to leave the data in a consistent state (i.e., as if they executed in some sequential order). However, if multiple **MPI_PUT** operations target the same memory location or if MPI-accumulate operations are mixed with **MPI_PUT** or **MPI_GET**, the resultant data is undefined. For AM-accumulate, no such interoperability semantics are defined. Unlike MPI-accumulate operations, the MPI implementation cannot keep track of the atomicity of AM-accumulate operations. Hence, MPI-accumulate interoperability semantics are not relevant for AM-accumulate.

IV. Generalized Active Messages

In this section we present our new design for MPI-interoperable AMs that addresses the shortcomings discussed in Section III. The design still leverages the MPI RMA framework but is no longer based on MPI-accumulate operations.

A. Generalized User-Defined Function Handlers

We first propose a new prototype of the user-defined function handler, **MPIX_AM_USER_FUNCTION**, that would execute when an AM arrives at a target (Figure 1), referred to here on as the "AM handler".

In the high-level working model the user defines a function handler with the **MPIX_AM_USER_FUNCTION** prototype and creates an MPI operation (**MPI_OP**) with that handler, using **MPIX_AM_OP_CREATE**. Once an MPI operation is created, the user collectively registers the operation across a group of processes where every process provides a functionally equivalent function handler.[1] Thus, the MPI operation represents a group of functionally equivalent handlers distributed across processes. For such collective registration, in our previous work, we had proposed the function **MPIX_AM_WIN_OP_REGISTER**, which we reuse here. The functional equivalence of the handlers makes it valid for the MPI implementation to replace one handler with another. For example, instead of transmitting the AM to the target process, the MPI implementation can fetch the target data locally and execute the AM using the local equivalent handler. Such an approach is particularly useful for shared-memory systems where the "remote data" might be directly visible to the process through shared memory.

The handler executes in user context (as opposed to an interrupt or signal context) and has access to three buffers: `input`, `persistent`, and `output`. Data in the `input` buffer is provided by the origin; data in the `output` buffer is created during the AM handler and is returned to the origin at the completion of the AM. Both the `input` and `output` buffers are private to the AM handler and are temporary. That is, neither the buffer nor its content is valid outside of the AM function handler. The MPI implementation can stage such data in temporary buffers and discard the buffers or the data in those buffers at the end of the AM handler. The `persistent` buffer points to the part of the target window that the AM handler has access to and is persistent across AMs. That is, the buffer is available and valid outside of the AM handler as well. The AM handler can update the data in the persistent buffer.

B. Active Message Triggers

We propose a new routine for issuing AMs, **MPIX_AM** (Figure 2), that manages the data and computation associated with an AM. This section describes the data associated with the AM. The computational function handler is represented by `am_op` and was described in Section IV-A.

MPIX_AM describes the computation to be performed on the data but does not specify where the computation actually occurs. Specifically, the AM origin and target only describe the locality of the data. The MPI implementation can choose to execute the computation on the target, origin, or any other location. While forcing the computation to always occur at the target would allow applications to better control the computational resource usage, it would take away the MPI implementation's capability to trade computational locality for less data movement, for example by moving the target data to the origin process and computing on it locally. Unfortunately, there is no clear winner between the two options. In our design, we allowed the MPI implementation to have more flexibility at the cost of fewer guarantees on the computational locality, but the alternative choice is also reasonable.

MPIX_AM allows the user to control parameters associated with three buffers: `input`, `output`, and `persistent`. The `input` buffer represents data that would be transmitted to the AM handler as input data. `origin_input_addr` provides the origin local buffer associated with the input data, while `origin_input_segment_count` and `origin_input_segment_datatype` represent the data layout. When the data is transferred to the target process, we allow the data representation to be modified to a different layout as represented by `target_input_segment_datatype`. This capability is useful for applications that use sparse data layouts on the origin for the input buffer (e.g., elements on the nonleading dimension of a matrix), but can represent them in a more space-concise format at the target (such as a contiguous list of elements). Note that the type signature of `origin_input_segment_datatype` does not

[1] Two handlers are functionally equivalent, if one can be executed instead of the other to get an equivalent result. For architectures that use different byte-widths or byte-ordering for datatypes, the MPI implementation will need to do the necessary data transformation before executing the handler.

```
MPIX_AM_USER_FUNCTION(input_addr, input_segment_count, input_segment_datatype, persistent_addr,
                      persistent_count, persistent_datatype, output_addr, output_segment_count,
                      output_segment_datatype, num_segments, segment_offset)
 IN     input_addr              address of input buffer (choice)
 IN     input_segment_count     number of elements in one input segment (non-negative integer)
 IN     input_segment_datatype  datatype of each entry in input segment (handle)
 INOUT  persistent_addr         address of persistent buffer (choice)
 INOUT  persistent_count        number of elements in persistent buffer (non-negative integer)
 INOUT  persistent_datatype     datatype of each entry persistent buffer (handle)
 OUT    output_addr             address of output buffer (choice)
 OUT    output_segment_count    number of elements in one output segment (non-negative integer)
 OUT    output_segment_datatype datatype of each entry in output segment (handle)
 IN     num_segments            number of segments in input and output buffers (non-negative integer)
 IN     segment_offset          current segment offset in input and output buffers (non-negative integer)
```

Fig. 1: Prototype of AM user-defined function

```
MPIX_AM(origin_input_addr, origin_input_segment_count, origin_input_segment_datatype, origin_output_addr,
        origin_output_segment_count, origin_output_segment_datatype, num_segments, target_rank,
        target_input_segment_datatype, target_persistent_disp, target_persistent_count, target_persistent_datatype,
        target_output_segment_datatype, am_op, win)
  IN   origin_input_addr               initial address of origin input buffer (choice)
  IN   origin_input_segment_count      number of entries in each segment in origin input buffer (non-negative integer)
  IN   origin_input_segment_datatype   datatype of each entry in origin input buffer (handle)
  OUT  origin_output_addr              initial address of origin output buffer (choice)
  IN   origin_output_segment_count     number of entries in each segment in origin output buffer (non-negative integer)
  IN   origin_output_segment_datatype  datatype of each entry in origin output buffer (handle)
  IN   num_segments                    number of segments in origin input and output buffers (non-negative integer)
  IN   target_rank                     rank of target process (non-negative integer)
  IN   target_input_segment_datatype   datatype of each entry in target input buffer (handle)
  IN   target_persistent_disp          window offset to target persistent buffer (non-negative integer)
  IN   target_persistent_count         number of entries in target persistent buffer (non-negative integer)
  IN   target_persistent_datatype      datatype of each entry in target persistent buffer (handle)
  IN   target_output_segment_datatype  datatype of each entry in target output buffer (handle)
  IN   am_op                           user-defined operation for the AMs (handle)
  IN   win                             window object used for communication (handle)
```

Fig. 2: Prototype of AM trigger routine

need to match that of `target_input_segment_datatype`. However, there must exist a non-negative integer 'N', where the type signature of `origin_input_segment_count` × `origin_input_segment_datatype` should match that of N × `target_input_segment_datatype`. In other words, the runtime system should be able to represent the data in each segment using a collection of `target_input_segment_datatype` elements. 'N' is internally calculated by the MPI implementation.

The `output` buffer represents data that is returned to the origin once the AM handler completes. `origin_output_addr` provides the origin local buffer associated with the output data, while `origin_output_segment_count` and `origin_output_segment_datatype` represent the data layout. Like the origin buffer, the data layout used at the target process can be different from that returned to the origin process, as represented by `target_output_datatype`. Each segment can be viewed as a unit of work to be executed in the AM. The input and output segment datatypes and their counts represent the data associated with each unit of work.

The `persistent` buffer represents data that already exists at the target process and is used within the AM. `target_persistent_datatype` and `target_persistent_count` represent the portion of the data that is accessed by the AM. While the AM can represent the entire target memory window using these parameters, an accurate representation of the required target data can allow the MPI implementation to optimize data movement in some cases (e.g., where the target data is smaller than the origin data, by fetching the target data and computing on it locally) or better identify opportunities for concurrency or out-of-order execution of AMs.

C. Data Streaming in Active Messages

Each AM comprises multiple segments as represented by the `num_segments` parameter. The MPI implementation is allowed to split an AM at any system-dependent size at the granularity of one segment. Such capability is useful, for example, when the user or the MPI implementation does not have enough temporary buffers to stage the entire input and output data specified by the user. Even when enough temporary buffer space is available, the MPI implementation can choose to pipeline the data transfer with the AM computation to improve performance. One noteworthy difference compared with MPI-accumulate operations is the granularity of segmentation. As described in Section III, in MPI-accumulate operations, the MPI implementation is permitted to segment a message at the granularity of predefined datatypes. With **MPIX_AM**, the minimum granularity of segmentation is user-defined.

D. Data Buffering Requirements

As described in Section IV-A, each AM handler is associated with a temporary `input` and `output` buffer that are valid only within the AM handler. An important question is who

is responsible for allocating and maintaining such temporary buffers. Most previous AM frameworks assume that the runtime system (in this case, MPI) would allocate and maintain such buffers. However, given that there is no upper bound on how much memory an AM would require, that is not a reasonable assumption and must be carefully defined.

To this end, we propose two new routines: **MPIX_AM_WIN_BUFFER_ATTACH** and **BUFFER_DETACH**. **BUFFER_ATTACH** allows the user to provide temporary buffer space to the MPI implementation to accommodate incoming AMs. **BUFFER_DETACH** reclaims the buffer from the MPI implementation. While the MPI implementation might also internally allocate additional temporary buffers, the application cannot assume the availability of such internal buffers.

The size of the user-provided buffer must be large enough to accommodate input and output buffers corresponding to at least one AM segment. Also, the user buffer is shared by AMs from all origins. Thus, the origin MPI implementation might need to perform appropriate synchronization with the target to "reserve" a part of the user-provided buffer before it can send the AM data. Furthermore, the user-provided temporary buffer must be large enough to accommodate the target input and output buffers in the user-described, potentially sparse, data layout. In other words, the temporary buffer should be at least the MPI true extent of the target input and output datatype counts (i.e., the extent returned by **MPI_TYPE_GET_TRUE_EXTENT**). While the interface allows sparse datatypes which have a small size but a large extent to be used in the AM handler, such datatypes are discouraged in practice because of the large memory usage they encompass.

E. Correctness Semantics

1) Ordering: We define three types of ordering: (1) between AMs with the same operation; (2) between AMs with different operations; and (3) between segments within one AM. By default, our framework imposes strict ordering for all three types from the same origin to the same target on the same window with overlapping target buffers. For all other cases, there is no ordering. The default strict ordering allows applications to reason about the state of the target window buffer when multiple AMs update it. The application is assured that a later AM is guaranteed to see the changes made by previous AMs. However, such strict ordering requirements also place a performance penalty on the MPI implementation. For example, if an AM is blocked because of lack of sufficient buffer space or any other issue, a future AM might also need to block.

To alleviate this issue, we allow the user to control the ordering of AMs using the MPI info hint **am_ordering** that is set on the MPI window during window creation. The value is a comma-separated list of the required ordering with permitted values **sameop**, **diffop**, and **sameam**, for the three cited types of ordering, respectively. The default value for **am_ordering** is "**sameop**,**diffop**,**sameam**," which imposes strict ordering for all three types. Any subset of these three orderings, or a value "**none**" can be specified in order to relax strict orderings. Reduced ordering guarantees can be beneficial for some applications, for example those which use AMs that only read the target data but do not update it. For such applications, the more relaxed semantics can allow the MPI implementation to reorder operations for better performance. We note that AMs are completely unordered relative to other MPI operations.

2) Concurrency: When one or more origin issues multiple AMs on the same target, the target can either serialize these messages or execute them concurrently. While concurrent execution has an obvious performance potential with respect to the amount of computational resources used, it is restrictive for applications since the AM handler has to be careful with respect to its data accesses and rely on atomic operations or locks in order to not conflict with other concurrent AMs. For some applications, such a model might not even be feasible. To handle this issue, by default, we require the MPI implementation to behave as-if the AMs are executed in some sequential order. An MPI implementation is free to apply AM operations concurrently for cases where concurrency is inconsequential. For example, if the implementation can prove to itself that the target data is nonoverlapping (on the granularity that it cares about), then it can execute them concurrently. For cases where the MPI implementation cannot easily identify whether such concurrency would be inconsequential, the MPI implementation might need to serialize the AMs at the target.

For applications that can handle concurrent AMs, the user can further provide a hint to the MPI implementation using an epoch start-time assertion, **MPIX_MODE_CONCURRENT_AM**, that would specify to the MPI implementation that within this epoch the AMs are concurrency-safe. For example, applications whose AMs do not overwrite each other's changes or only read data from the target window can provide such an assertion. If such an assertion is passed, AMs that are within the same epoch and are not separated by **MPI_WIN_FLUSH** operation (or other flush-style operations) may be executed concurrently on a window at the same target. AMs from the same origin or from different origins may be executed concurrently.

We note that an MPI implementation might not be able to control the ordering of concurrent AMs. Thus, if the user requests strict ordering but provides an assertion to specify that the AMs are concurrency-safe, an MPI implementation might still have to disable concurrency to meet the ordering requirement. We also note that concurrency of AMs or the lack thereof does not restrict AMs from executing concurrently with other RMA operations.

3) Atomicity: Our framework does not guarantee atomicity of updates between concurrent AMs or between AMs and other RMA operations. Thus, if two AMs update the same memory byte on a target, the resultant value is undefined. Similarly, if an AM accesses the same memory region as another RMA operation, the resultant value is again undefined. An exception to this rule is read-only accesses; if multiple AMs read from the same location, or if an AM and another RMA operation read from the same location, such accesses would return the actual value at the target memory. We note that the atomicity requirements described above are valid only for the `persistent` buffer used in the AM and do not impact the `input` and `output` buffers that are private to each AM.

4) Memory Consistency: MPI RMA defines two memory models: UNIFIED and SEPARATE. In the SEPARATE model, the MPI process can be viewed as having two copies of the window: the *public* window that is addressable by all processes and the *private* window that is local to each process. In the UNIFIED model, there is a single copy of the window. In practice, the SEPARATE model is more natural for non-cache-coherent architectures where the consistency of the cache with respect to memory has to be handled in software by the MPI implementation, while the UNIFIED model is more natural for cache-coherent architectures where such consistency is managed in hardware. RMA operations access the *public*

window, and local loads/stores access the *private* window. One primary difference between AMs and traditional RMA operations is that AM handlers access the *private* window rather than the *public* window. The reason is that operations involved in the user function are local loads/stores invoked by the target process and are not puts/gets/accumulates invoked by the origin process. Furthermore, concurrent AMs at the same target might each have a separate private window.

This difference between AMs and traditional RMA raises several subtle interoperability issues. For example, in the SEPARATE model, if an AM and a regular RMA operation update the same window, the state of the data in that window is undefined even if these operations update nonoverlapping memory regions. The reason is that during an AM, if the target process fetches a block of data to cache and an RMA operation updates another nonoverlapping variable on same cache line, such an update would be lost when the cache line is written back to memory. MPI cannot keep track of such accesses. Similarly, if two concurrent AMs access nonoverlapping regions on the same target window, they both might have two different copies of the private window, and can thus overwrite each other's changes to the window memory. Further, in both memory models, each AM has to ensure that it sees updates issued by previous RMA operations and leaves the window in a consistent state for future RMA operations. Since the AM handler function computes on the private memory region, in the SEPARATE model, the MPI implementation will need to flush the cache back to memory before returning the AM completion notification to the origin. In the UNIFIED model, while the status of the cache is managed by hardware, the MPI implementation will still need to perform a full memory barrier before and after the AM function handler executes, in order to ensure that future reads from the window memory return the updated data.

F. Other Considerations

One additional aspect to consider is whether an AM handler can call other MPI functions. Two factors must be considered in this regard. First, an AM handler might be executed by the MPI implementation while it is making progress for another MPI function (e.g., while it is waiting for a request to complete). In this case, allowing the AM handler to execute an MPI function would result in the execution reentering the MPI stack, thus requiring the MPI routines to be reentrant safe. Second, since an AM might execute concurrently with the main application thread, calling MPI routines within an AM would require MPI to be initialized in a thread-safe manner. Since the application does not know whether the MPI implementation would execute the AM concurrently with the application thread, the application would always need to initialize MPI with a higher thread-level than it would otherwise require. Alternatively, since the MPI implementation would not know whether the application is planning to make MPI calls within the AM handler or not, it would automatically have to promote the thread-level to a higher value than what the application would otherwise require. To avoid these issues, in our current model we do not allow AM handlers to call other MPI functions.

V. Evaluation

For our evaluation, we used a 310-node system, with each compute node consisting of 16 cores (total of 4,960 cores). The nodes are connected with QLogic QDR InfiniBand Interconnect (fat-tree topology). Our implementation is based on MPICH-3.0.2. We implemented two common operations using AMs. The first operation is a remote search operation, where the origin initiates AMs with string sequences (20 characters) to the target to search for matching strings and return them to the origin. The second operation is a remote compute of the summation of absolute values of two arrays. In the first operation each segment has 20 characters as input and 20 characters as output, while in the second operation each segment has 100 integers as input and 100 integers as output. All experiments use an internal system buffer of 8 KB per peer process, except Figures 4 and 5 which vary the buffer size.

A. Streaming Active Messages

As discussed in Section IV, **MPIX_AM** is designed to allow the MPI implementation to split an AM into smaller segments for better pipelining or to limit memory usage for temporary buffers. Here, we analyze the impact of such streaming (or pipelining) of segments within an AM. Figure 3(a) illustrates an experiment with the *search* operation, where we measure the latency of performing a single AM within an epoch. When the epoch ends, the AM has completed and a completion notification has been sent to the origin process. The AM consists of 100 segments, each segment requiring 20 bytes for each of the input and output buffers. The MPI implementation uses an internal buffer of 8 KB per peer process; thus, with a single AM within the epoch, we are guaranteed that the MPI internal buffer can accommodate the entire AM and the user buffer is never used.

We notice a continuous drop in latency as we increase the size of each pipeline unit (legend "latency"). With a pipeline unit of 10 segments the latency of each AM is 115μs, while with a pipeline unit of 20 segments the latency drops to 95μs, a 17% reduction. Further increasing the pipeline unit drops the latency by another 16%. To analyze this behavior, we profiled the execution by measuring the cost of just the computation without the overhead of AM data transfers and synchronization (legend "function call time"). The drop in computation time with increasing pipeline unit size is due to the reduction in the number of function calls. When each pipeline unit is 10 segments long, the AM handler is invoked 10 times, each with 10 segments to compute. When each pipeline unit is 100 segments long, the AM handler is invoked just once with 100 segments to compute. While the total amount of computation in both cases is the same, the former has a 10-fold larger number of function invocations. We also notice that the time to execute the computation itself closely follows the trend of the AM latency. The additional overhead compared to the computation time is attributed to the AM data transfer and synchronization.

Figure 3(b) illustrates a similar experiment with the *search* operation, where we measure the throughput of an AM by performing 100,000 AMs within an epoch. Each AM consists of 100 segments and each segment requires 20 bytes for each of the input and output buffers. The MPI implementation uses an internal buffer of 8 KB per peer process; thus AMs use the internal buffers when available and fall back to the user buffer when internal buffers are in use. We can seen that when the pipeline unit is 40 segments, the highest throughput is achieved. To analyze this behavior, we profiled the number of segments that utilize the MPI internal buffers vs. the user buffers (also Figure 3(b)). As we increase the pipeline unit, an increasingly large fraction of the segments uses the user buffer, thus requiring additional synchronization with the target. At the same time, for very small pipeline unit sizes, the number

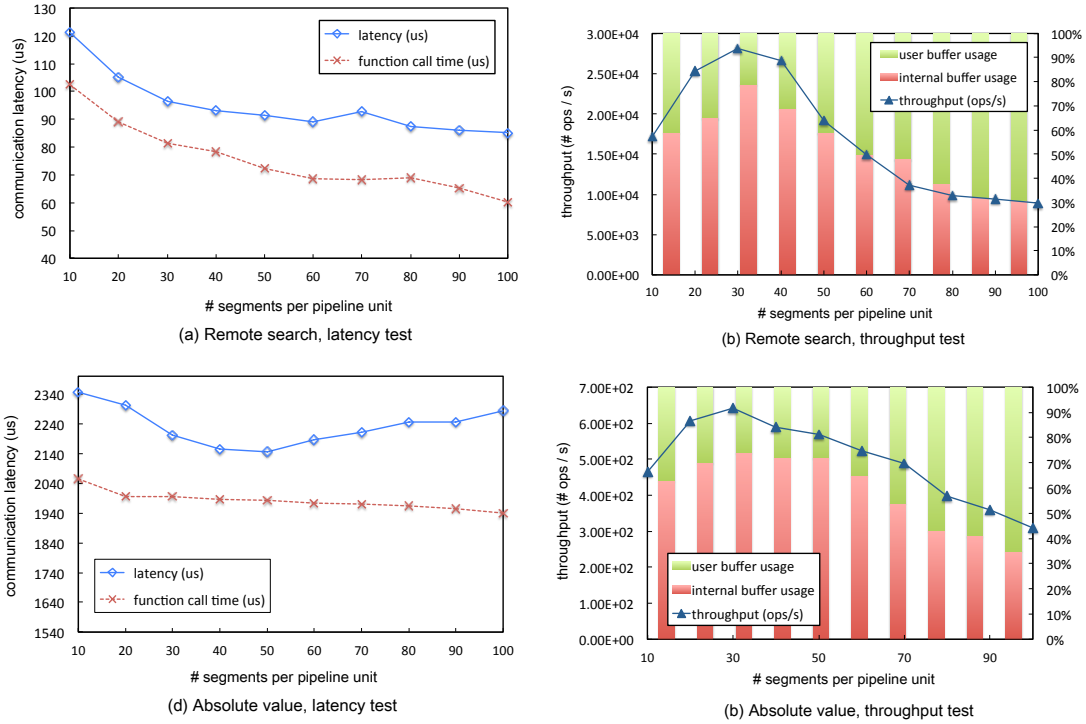

Fig. 3: Communication latency and operation throughput with different numbers of segments per AM packet

of function calls can add a high overhead, as illustrated in Figure 3(a). Consequently, we expect the best throughput to be somewhere in between. For the *search* operation, a pipeline unit of 40 segments happened to be the sweet spot, although this value depends on the computational operation and other system characteristics.

Figures 3(c) and (d) show the results of experiments similar to those above but for the absolute value computation. The performance trends are similar to those in Figures 3(a) and 3(b) except that in Figure 3(c) the AM latency reaches its lowest at a pipeline unit size of 40 segments and increases after that. The reason is that, unlike the search computation, the absolute value computation is more input data intensive, requiring a larger amount of data to be transferred between the origin and target. Thus, the pipeline size makes a significant difference in how well the computation is overlapped with data movement.

B. *Impact of Internal Buffers*

The user is required to attach temporary buffers that can be used by the MPI implementation to stage AM input and output data. However, the MPI implementation can choose to allocate additional internal buffers to improve performance in some cases. In our implementation, the target allocates a small amount of buffer space statically assigned to each origin process (total internal buffer size would be this size times the number of processes in the system). In this section, we evaluate the impact of such internal temporary buffers on the performance of AMs.

Figure 4 demonstrates the impact of increasing internal buffer size on the throughput of AMs. We used the *search* benchmark for our experiments, with each AM containing 100 segments and each epoch issuing 100,000 AMs. We notice that the throughput of the AMs generally increases with increasing internal buffer size up to a point and then levels off. The reason for the performance increase is straightforward: when more internal buffer space is available, the MPI implementation

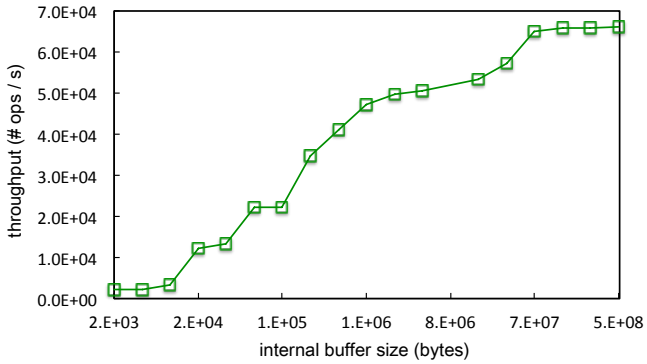

Fig. 4: Throughput: Impact of System Buffer Size

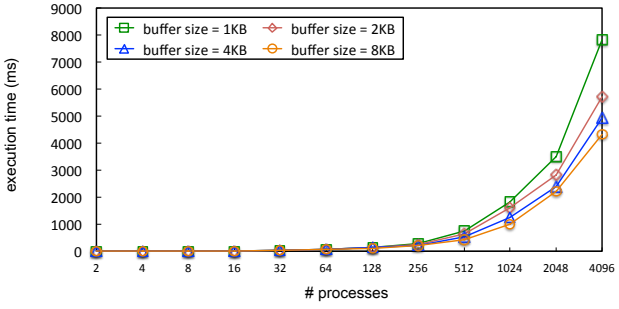

Fig. 5: Execution Time: Impact of System Buffer Size

can directly use the internal buffer instead of performing a handshake with the target to get access to the shared user buffer. With increasing internal buffer space, the number of times such a handshake occurs reduces, improving performance. The reason for the leveling off of the performance for very large internal buffer sizes is subtle: the amount of internal buffer size that a series of AMs can utilize is limited. As the MPI implementation continues to issue more AMs, previously

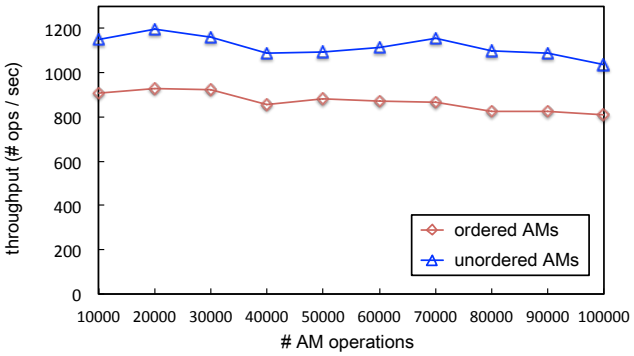

Fig. 6: Throughput: Impact of Ordering

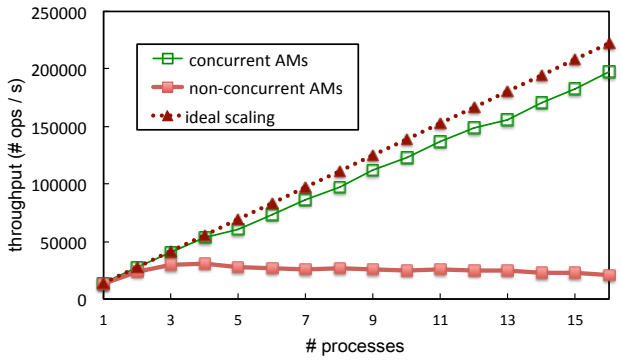

Fig. 7: Throughput: Impact of Concurrency

issued AMs complete execution, thus freeing up buffers. Even with a large number of AMs, the runtime system can reach a steady-state where the data transmission and AM computation would match up and more temporary buffer space would not result in a performance increase.

Figure 5 demonstrates the impact of the internal buffer size when multiple origins issue AMs to the same target. This experiment is similar to the previous experiment except that a large number of origins issue AMs to the same target. We notice that for a small number of origins, the internal buffer size (different lines on the graph) does not make much difference. However, as the number of processes grows, we notice as much as a 1.7 times difference in performance.

C. Impact of Ordering

In this section, we analyze the impact of ordering between AMs on performance. In our experiment, we issue a number of AMs within a single epoch alternating between ones that requires a large temporary buffer space and ones that require a small temporary buffer space. When the user requires strict ordering between the AMs, the MPI implementation is required to finish all previous AMs before issuing the next one. More specifically, if an AM is blocked waiting for more memory than what the MPI internal buffer can offer (thus requiring the user buffer), all future AMs will also be blocked even if they can fit into the MPI internal buffers. When the user does not require such ordering, we can issue later AMs early, while waiting for the larger user buffer to be available. Figure 6 illustrates the performance difference achievable by such lack of ordering. Removing strict ordering requirements can provide close to 25% improvement in performance. On architectures such as Blue Gene, where multiple paths exist between the origin and target, the ability to issue AMs out of order can also allow the MPI implementation to use multiple paths for the operations, thus further improving performance.

D. Concurrent Active Messages

In this section, we analyze the impact of executing AMs concurrently. In our experiment, we have a number of origins issuing AMs on a common target on the same physical node. When there is no concurrency in the AMs, all AMs are forwarded to the target and executed. However, when the user allows for AM concurrency, each origin can take advantage of the fact that the window data is shared across processes and compute directly at the origin process itself. Figure 7 illustrates the performance achievable through concurrency on an 16-core system. Without concurrency, when number of processes is 16, the aggregated throughput achieved by the AMs can be more than nine-fold worse than with concurrency enabled. With larger numbers of cores, we expect this difference to grow.

VI. Conclusions

We presented MPI-interoperable generalized AMs. We analyzed usage restrictions and performance disadvantages when extending MPI accumulate operations to support AMs, including limitations on data layouts, data access, and memory inefficiency issues. Based on the analysis, we proposed a new design for MPI-interoperable AMs to provide a more flexible and general usage model.

References

[1] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *ISCA*, 1992.

[2] X. Zhao, D. Buntinas, J. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp, "Towards Asynchronous and MPI-Interoperable Active Messages," in *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2013.

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

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

[5] D. Bonachea, "GASNet specification, v1.1," University of California, Berkeley, Tech. Rep. CSD-02-1207, October 2002.

[6] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer," in *International Conference on Supercomputing (ICS)*, 2008.

[7] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with LAPI - a new high-performance communication library for the IBM RS/6000 SP," in *International Parallel Processing Symposium*, 1998.

[8] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "PAMI: A parallel active message interface for the Blue Gene/Q supercomputer," in *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2012, pp. 763–773.

[9] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[10] D. Bonachea, "AMMPI: Active messages—over MPI - quick overview," http://www.cs.berkeley.edu/~bonachea/ammpi/.

[11] J. Meng, J. Yuan, J. Cheng, Y. Wei, and S. Feng, "Small World Asynchronous Parallel Model for Genome Assembly," *Springer Lecture Notes in Computer Science*, vol. 7513, pp. 145–155, 2012.

[12] F. Xia and R. Stevens, "Kiki: Massively parallel genome assembly," https://kbase.us/, 2012.

[13] R. J. Harrison, "MADNESS: Multiresolution ADaptive NumErical Scientific Simulation," https://code.google.com/p/m-a-d-n-e-s-s/, 2003.