Optimization Strategies for MPI-Interoperable 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, such as those in bioinformatics and social network analysis, differ from traditional scientific applications in that they often involve data-driven and irregular computation/communication patterns, making them ill-suited for traditional data movement approaches. Active Messages (AM) is an alternative programming model that allows dynamically moving computation closer to data, rather than moving the data to the local process. In our previous work, we proposed an MPI-interoperable AM framework that allows existing MPI applications to incrementally take advantage of AM capabilities. While that work presented a baseline implementation of how AMs semantically interact with the rest of the MPI infrastructure, it had several performance shortcomings. In this paper, we analyze these performance shortcomings and propose three optimization strategies: one implicitly derived by the MPI implementation and two explicitly hinted to by the application user. In addition to the detailed description of these optimization strategies, the paper presents a thorough performance evaluation on a 4096-core cluster that demonstrates considerable performance advantages from these strategies.**

Keywords*—*Active messages; MPI; Data-intensive applications; RMA; Multicore

I. Introduction

Data-intensive applications, such as graph algorithms in social network analysis and genome assembly applications in bioinformatics, differ from traditional scientific applications in that they often involve data-driven and irregular computation/communication patterns. Traditional programming models that are designed for regular and structured applications are not well suited for such applications. Alternative programming models are desirable. The Active Messages (AM) paradigm [1], proposed by von Eicken in 1992, is an alternative parallel programming paradigm that can be more natural for such applications. With AMs, the sender of a message specifies a message handler to be executed at the receiver upon arrival of that message. When the message arrives, the corresponding handler is triggered to process data in that message. Compared with the traditional SEND/RECV and PUT/GET models that move the data closer to the computation, the AM model moves computation closer to the data.

The Message Passing Interface (MPI) [2] is the de facto standard for parallel programming on large-scale systems and is available on virtually every system in the world. Given the popularity of MPI and the importance of the AM paradigm for data-intensive applications, in our previous work [3], [4] we proposed a generalized framework for MPI-interoperable AMs by leveraging the MPI remote memory access (RMA) framework. We presented a detailed set of functionality and semantics that provides generally usable AMs that are compatible with the MPI-3 standard.

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.

While our previous work presented a baseline implementation of how AMs semantically interact with the rest of the MPI infrastructure, the implementation had several performance shortcomings. For example, the semantics of the AM framework allowed users to provide buffers for AM processing that could be used by different processes initiating AMs on a particular target process. Because of the shared nature of these buffers, however, considerable performance was lost in synchronization and coordination overheads, especially on large-scale systems. Furthermore, for highly irregular applications where the amount of data associated with the AMs is highly variable, our baseline implementation resulted in a significant amount of additional data being transferred, causing additional performance loss.

In this paper, we examine these performance shortcomings through a detailed analysis of the runtime behavior of the MPI infrastructure. Our analysis demonstrates large stalls and idleness during synchronization, which increase with system size. The analysis also indicates that a significant amount of unnecessary data is transferred for highly irregular applications. To address these shortcomings, we propose three optimization strategies. The first strategy is an implicit optimization that takes advantage of application synchronization in order to avoid additional internal synchronization and transparently improve performance. The second and third strategies are more explicit. The second strategy uses an application hint to learn additional application semantic information in order to further reduce synchronization overheads in some cases. The third strategy uses a new "vector-based" AM function that allows highly irregular applications to better describe their data layout, thus reducing the amount of data the implementation has to transfer.

In addition to the detailed description of these optimizations, this paper presents a reference implementation and a thorough performance evaluation of the proposed strategies on a 4096-core InfiniBand cluster. Our evaluation demonstrates a considerable performance advantage from the proposed techniques, thus verifying their validity and applicability in a generalized and MPI-interoperable AM framework.

For readability, 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_**. This paper heavily relies on the semantics of the MPI-3 RMA model. Since we cannot provide details of these semantics because of space restrictions, we highly recommend that the reader of this paper also read the following literature to gain a better understanding of these semantics: [5], [6].

II. Overview of MPI-Interoperable AMs

In [4], we proposed a generalized framework for MPI-interoperable AMs that allows existing MPI applications to incrementally benefit from AM functionality. We defined the semantics and presented correctness definitions of such a

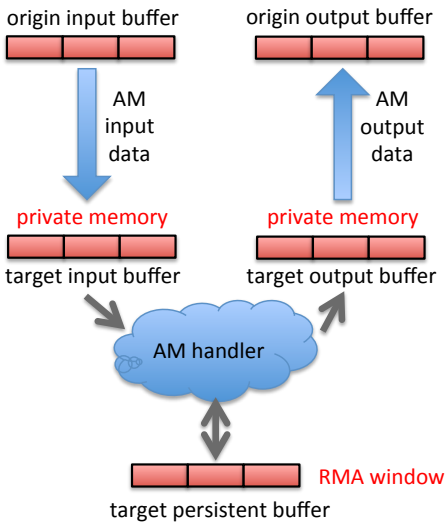


Fig. 1: Workflow in the generalized AM framework

framework, including memory consistency, atomicity, ordering, and concurrency semantics. The workflow of this AM framework is illustrated in Figure 1. We proposed a new routine, **MPIX_AM**, for issuing AMs. Using this routine, application programmers can manage data content and layout in five associated buffers: origin input buffer, target input buffer, target persistent buffer, target output buffer. and origin output buffer. As shown in the workflow, the origin input data is sent to the target and is staged in the temporary *target input buffer*. This temporary staged data serves as the input to the AM handler on the target, which in turn stores its generated output into the temporary *target output buffer*. Once the AM handler completes, the output data is returned to the *origin output buffer*. The *target persistent buffer* represents data that already exists at the target process's public window and is accessed within the AM handler. Unlike the target input and output buffers, the target persistent buffer retains its content after the AM handler completes. Thus, all updates on this buffer can be seen by future MPI RMA and AM operations. We also proposed a new prototype for the AM function handler, **MPIX_AM_USER_FUNCTION**, that is invoked at the target upon arrival of the corresponding AM.

Since the target input and output buffers are temporary buffers that are private and valid only within the AM handler, an important question is who is responsible for the allocation and management of these temporary buffers. Most AM frameworks assume that the runtime system will manage such buffers. However, since there is no upper bound on the buffer space needed for an AM, such an assumption is impractical. In our AM framework, we proposed two new routines, **MPIX_AM_WIN_BUFFER_ATTACH** and **MPIX_AM_WIN_BUFFER_DETACH**, to allow the user to provide appropriately large temporary buffers to the MPI runtime that it can use to stage AMs. These user buffers are shared by all origin processes, thus requiring each origin to synchronize with the target in order to coordinate on the buffer usage.

One additional concept from our previous work that the reader should be familiar with is that of "segments." For a large AM, the MPI runtime system may choose to split it into smaller pipeline units. Without additional information from the user, however, the MPI implementation cannot tell what an appropriate granularity would be. For example, consider the Kiki genome assembly application [7]. In this application, the user has a bunch of query sequences that need to be searched on a remote dataset. The user can create a single AM with all query strings, thus giving the MPI implementation the flexibility to split it into smaller pipeline units. However, the MPI implementation must be careful to make sure that each pipeline unit contains an integral number of strings, since the application cannot search for a partial string in the dataset. To allow for such capabilities, we use the concept of segments, which represents the minimum granularity of splitting AMs. In the case of Kiki, each string would form a segment, while the AM could have several hundreds of segments. We note that while a segment is a user-defined concept, the pipeline unit used is internal to the MPI implementation and can be system specific.

Segmentation of AMs is useful in many cases. For example, the MPI implementation can use this information to pipeline an AM so that the data movement and computation are overlapped for better performance. Further, when there are not enough buffers for the entire AM data to be staged at the target, the MPI implementation can split the AM into smaller pipeline units that fit into the available buffers. Note that in a correct application the user must provide enough user buffers to accommodate at least one input and output segment of the AM.

III. Performance Shortcomings of MPI-Interoperable AMs

In this section, we analyze the performance shortcomings in the existing base implementation of MPI-interoperable AM framework. Based on this analysis, we propose several optimization strategies in Section IV.

A. Synchronization Stalls in Data Buffering

As described in Section II, the semantics of our AM framework require the user to ensure that enough temporary buffers are attached to the window so that the input/output data corresponding to the AM can be accommodated at the target. The MPI implementation can, however, provide additional system internal buffers to improve performance.

Such internal buffers can be managed in multiple ways. For instance, a large chunk of memory can be shared among all origin processes. In such a design, each origin process needs to coordinate with the target to "reserve" a part of the buffer before it can initiate an AM. Another possible design is to statically partition the buffer between the origin processes, so every origin process gets exclusive access to a part of the system buffer. The advantage of this approach is that since the buffer associated with each origin process is exclusive, no additional handshake is required between the origin process and target process in order to use the buffer. The disadvantage, however, is that such static partitioning reduces the amount of buffer space available to each origin process. Several other design possibilities exist where one could dynamically manage and adjust the amount of exclusive buffer space available to each origin process at runtime. In our implementation, we have not investigated all possible design options. Instead, we chose the second option; that is, we statically partition the system buffer to give each origin process exclusive access to a part of the buffer.

Irrespective of which design is chosen for internal system buffer management, we emphasize that these system buffers are limited. For large AMs or when a large number of outstanding

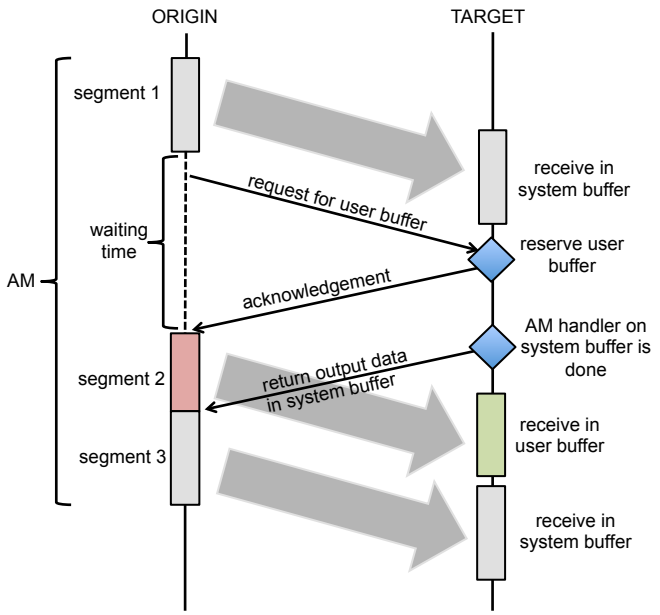


Fig. 2: Handshake operation for reserving user buffers

AMs are issued, the system buffer will eventually run out, and the MPI implementation will need to start using the user-allocated buffer. Given the shared nature of these user buffers, however, each origin process must perform a "handshake" with the target process in order to reserve some space in the user buffer before it can send its AM.

The overall handshake protocol is illustrated in Figure 2. In this example, consider an AM that has a large number of segments. The first few segments that can fit into the system internal exclusive buffers are sent immediately. Once these system internal buffers are exhausted, however, the origin process needs to send a handshake message in order to reserve space in the shared user buffer before it can send the next segment, thus idly waiting for buffer space at the target to become available. Depending on what fraction of time the origin process spends waiting, the performance of the AM framework can be substantially impacted. Issuing multiple nonblocking operations does not alleviate this issue, since the origin process can send only as much data as it knows the target can accommodate.

B. Inefficiency in Data Transmission

A common characteristic of several irregular data-intensive applications is that the amount of data returned by an AM is data-dependent. In bioinformatics genome assembly applications such as Kiki, where an input query string is searched on remote datasets, the amount of data returned depends on how many matches the AM handler can find in the remote dataset. Such information, unfortunately, cannot be predetermined easily. Thus, the usage model of MPI-interoperable AMs in such applications involves the application allocating a large local output buffer and issuing AMs that return data into this buffer.

Such a model has two obvious deficiencies. First, the amount of buffer allocated for output can potentially be large. Second, with the current **MPIX_AM** semantics, the AM handler cannot specify a different amount of output data size for each segment; thus, the amount of data returned to the origin process is equal to the total amount of buffer space allocated (i.e., the maximum AM output size). Obviously, this situation can be highly wasteful for irregular applications where the amount of data returned can vary significantly between AM segments.

IV. Optimization Strategies for MPI-Interoperable AMs

Based on the performance shortcomings described in Section III, we present here three optimization techniques for the MPI-interoperable AM framework.

A. Autodetected Exclusive User Buffers

The first optimization we propose takes advantage of application synchronization to reduce the internal synchronization required for user buffer management. As defined in MPI-3, RMA (or AM) operations can be issued only inside an epoch, either passive or active. For passive target epochs, an application first issues an **MPI_WIN_LOCK** to the corresponding target, followed by a bunch of AMs, and then closes the epoch with an **MPI_WIN_UNLOCK**. Such an epoch can be initiated in "exclusive" or "shared" mode. If an origin process acquires an exclusive lock at a target window, no other origin process can get either an exclusive or shared lock at the same target window. If an origin process acquires a shared lock at a target window, other origin processes can get a shared lock on the same target window, but not an exclusive lock.

Our proposed optimization strategy takes advantage of this model by internally keeping track of the lock acquisition status of each window. Thus, if an origin process has acquired an exclusive lock at a target window, by definition it is the only process that can access the target window and consequently the attached user buffers. In this scenario, we need to send only one synchronization message right after **MPI_WIN_LOCK** to fetch information on the target user buffers. For all subsequent AM operations, no more synchronization messages are needed. Note that this optimization is transparent to the user.

A corner case that we need to handle is detachment of user buffers. Applications are allowed to attach and detach an arbitrary number of buffers to a window dynamically. Before they can detach a user buffer, however, they need to ensure that no AMs are currently executing. Thus, an example such as the one illustrated in Figure 3 is a valid program. Notice that in the program the amount of user buffer space attached to the target window has changed while inside the exclusive lock epoch. That is, the AMs issued from lines 9–11 have access to both **user_buf_1** and **user_buf_2** that are attached to the target. With appropriate synchronization, however, the target can detach **user_buf_1**, leaving the later AMs on lines 20–21 with access only to **user_buf_2**. In such cases, our optimization of querying for the available user buffer space just once at the start of the epoch would no longer be correct. To handle this scenario, we resynchronize the user buffer information after every synchronization operation, such as an **MPI_WIN_FLUSH**. Such resynchronization, however, can result in loss of performance in cases where the user buffer was not detached at the target. Unfortunately, the MPI implementation cannot easily detect this automatically. To alleviate this issue, we allow users to pass a hint to the MPI implementation at window creation using the **MPI_Info** key **am_buf_interleave_am_detach**. The default value of **true** means that the user can interleave AM operations with **MPIX_AM_WIN_BUFFER_DETACH** operations, thus requiring additional synchronization as described above. By setting this value to **false**, however, the user can guarantee that **MPIX_AM_WIN_BUFFER_DETACH** operations

```
if (myrank == 0) {
    /* Barrier to ensure that buffers are attached */
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);

    /* AMs should have access to both user buffers */
    MPIX_Am(...);
    MPIX_Am(...);
    MPIX_Am(...);

    MPI_Win_flush(1, win);

    /* Barrier to inform that flush has completed */
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    /* AMs should have access to one user buffer */
    MPIX_Am(...);
    MPIX_Am(...);

    MPI_Win_unlock(1, win);
}
else if (myrank == 1) {
    MPIX_Am_win_buffer_attach(user_buf_1, 100, win);
    MPIX_Am_win_buffer_attach(user_buf_2, 200, win);

    /* Barrier to ensure that buffers are attached */
    MPI_Barrier(MPI_COMM_WORLD);

    /* Barrier to inform that flush has completed */
    MPI_Barrier(MPI_COMM_WORLD);

    /* detach user buffer */
    MPIX_Am_win_buffer_detach(user_buf_1, win);
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Fig. 3: Buffer detach example

are never interleaved with AM operations, thus requiring no additional synchronization. In such cases, the additional handshake operation in **MPI_WIN_FLUSH** can be eliminated.

B. User-Defined Exclusive User Buffers

The second optimization we propose is for cases where the MPI implementation cannot automatically detect exclusivity of user buffer access, such as with shared locks or MPI active target synchronization modes. In some cases, the application can algorithmically determine the maximum target buffer size that would be used by other origin processes and thus can determine the amount of target buffer space available to a given origin process. In such cases, if the application can pass this information down to the MPI implementation as an **MPI_Info** hint during window creation, the implementation can use this information to potentially reduce synchronization stalls.

In our proposed approach, we use a target-specific info key (**am_user_buf_**<**rank**>); the info value specifies the byte size of the user buffer space on that target that is guaranteed to be available. Note that this value specifies only the guaranteed buffer space and hence is necessarily conservative. More user buffer space might be dynamically available to the MPI implementation, which it can query for, using its handshake protocol. For AMs that fit in the "exclusive user buffer" space, however, no further handshake is required.

C. Improving Efficiency in Data Transmission

As described in Section III-B, with **MPIX_AM**, the AM handler cannot specify a different amount of output data size for each segment. Thus, a fixed amount of output data (equal to the maximum AM output size) is returned to the origin process at the completion of the AM. To alleviate this issue, we propose a new function for vector-based AMs (**MPIX_AMV**) and an associated AM handler prototype (**MPIX_AMV_USER_FUNCTION**). These functions are referred to as "vector" versions of the original **MPIX_AM_USER_FUNCTION** and **MPIX_AM** functions because one new vector argument, `output_segment_counts`, is added to them. This argument is an integer array of length `num_segments` with each entry indicating the count of elements in the corresponding output segment. For the AM handler, the MPI runtime allocates the `output_segment_counts` array, but how much data is actually generated needs to be filled by the handler function.

a) Output Data Layout: One design choice associated with vector-based AMs is how much buffer space must be allocated for the origin output buffer and how data is laid out in this buffer. Since the amount of data that will be generated is unknown, the user cannot know the buffer space required. Thus, we still require the origin output buffer to be large enough to fit the maximum data size returned by the AM.

With respect to the data layout of the origin output buffer, however, the most intuitive approach would be to place the entire output data in a contiguous segment of the output buffer. While convenient for the user, such a data layout has several performance shortcomings, particularly with respect to out-of-order execution of AM segments, as illustrated in Figure 4. For example, suppose the AM has four segments. As shown in Figure 4(a), if the latter two segments in the AM execute and return data earlier, the origin process cannot know at what offset it needs to place this output data, since it does now know how much output would be returned by the first two segments. In such cases, the MPI implementation has to either buffer out-of-order data or place it in the user buffer and reorder it once all of the data is available. Both options are expensive for performance. Arguably, to prevent this complexity, users can impose strict ordering between AMs, but that would sacrifice the concurrency of out-of-order AMs and the associated performance improvement.

Consider an alternative model where the output data is not placed in a contiguous buffer, but each segment is placed at a fixed offset calculated based on the maximum output size that each segment can generate (Figure 4(b)). This model might be slightly more inconvenient to the user, but the performance potential of this approach is much higher. In particular, since the location of each segment's output data is predetermined, no additional buffering or reordering is required. Data can be placed at the right location as soon as it arrives. For our framework, we chose this approach.

b) Data Packing vs. Data Transmission: In **MPIX_AMV**, because each segment can generate an uneven amount of data, the generated output data can be noncontiguous in memory both at the target within the AM handler and at the origin process. Thus, the MPI implementation at the target would need to consolidate this data into a temporary packing buffer in order to send it to the origin process, which in turn would unpack the data into the origin output buffer. In contrast, with **MPIX_AM**, since the amount of data generated is predetermined, the communication is often from contiguous memory.

The difference between the packing strategy that **MPIX_AMV** uses and the complete data movement strategy that **MPIX_AM** uses can be significant, in favor of **MPIX_AMV**, when the amount of actual data generated is much less

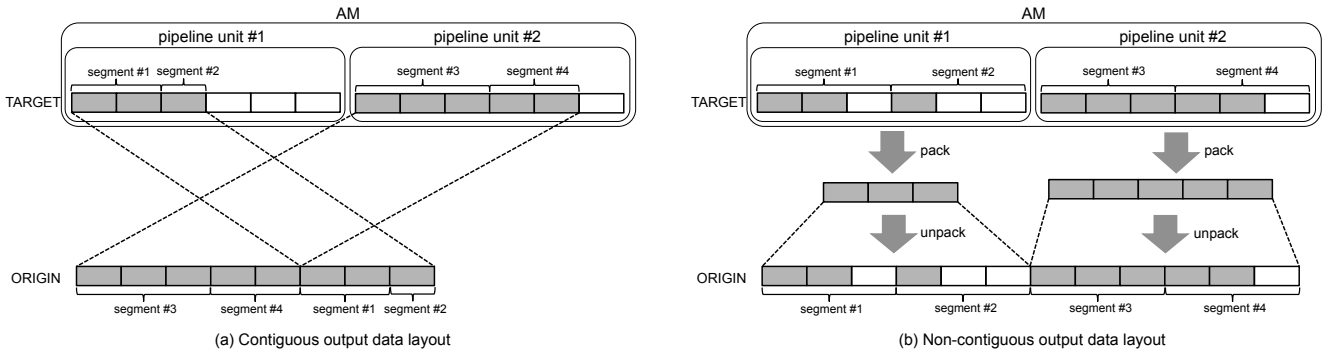


Fig. 4: Different strategies of origin output data layout

than the maximum buffer size. In such cases, packing and unpacking a small amount of data can be significantly faster than communicating a large amount of data. However, as the amount of data generated by the AM handler increases (as percentage of the maximum buffer size), this difference vanishes, and the packing overhead starts to dominate. To overcome this issue, our framework internally maintains a system-specific threshold for when to pack data and when to just transmit the entire data. When the amount of data is below this threshold, we pack and send the data. When above this threshold, we transmit all the data, including garbage data in the buffer that was not generated by the AM handler.

Note that in both strategies the entire count array of the output lengths is transmitted to the origin process. Thus the data transmitted will be slightly higher than what **MPIX_AM** would transmit when the handler generates the maximum amount of data. In such cases, **MPIX_AM** would be a better choice.

V. Evaluation

For our evaluation, we use a 310-node system, with each compute node consisting of 16 cores (total of 4,960 cores). The nodes are connected with Intel/QLogic QDR InfiniBand. Our implementation is based on MPICH-3.1b1. We use two types of AM operations. The first one is a *remote search* operation, where the origin initiates AMs with string sequences in order to search for matched string sequences in a remote dataset and return them to the origin. This is the most common operation in genome assembly applications such as Kiki and SWAP [8]. The second operation is a remote computation of the summation of *absolute values* of two arrays. In the first operation each segment consists of 20 characters (1 sequence) as input and 20 to 200 characters (1 to 10 sequences) as output (experiments in Section V-A return 1 sequence per segment, and experiments in Section V-B return multiple sequences per segment). In the second operation each segment contains 100 integers as input and 100 integers as output. All experiments use an internal system buffer of 8 KB per peer process. In experiments other than those shown in Figures 5 and 8, each process attaches 32 MB of user buffer.

In Section V-A, we compare the performance of the first two optimization approaches (*excl-lock-opt-impl* for autodetected exclusive user buffers and *win-opt-impl* for user-hinted exclusive user buffers) with a base implementation that does not take advantage of either optimization (*base-impl*). In Section V-B, we compare the performance of **MPIX_AM** with that of the newly proposed vector-based AM, **MPIX_AMV**. Because of space restrictions, we do not present *absolute value* results in Sections V-A2, V-A3, and V-A4. They have performance trends similar to those of the *remote search* operation.

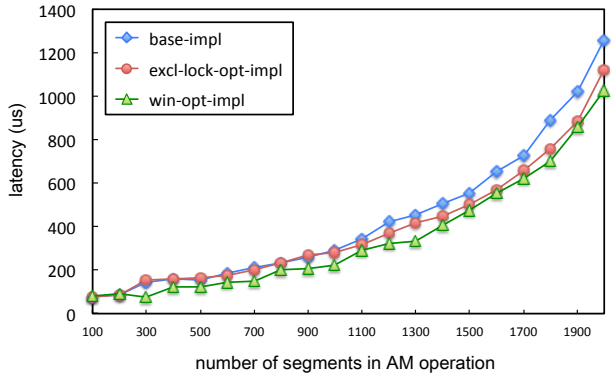
A. Effect of Exclusive User Buffers

We focus here on four effects: communication latency, operation throughput, scalability, and network contention.

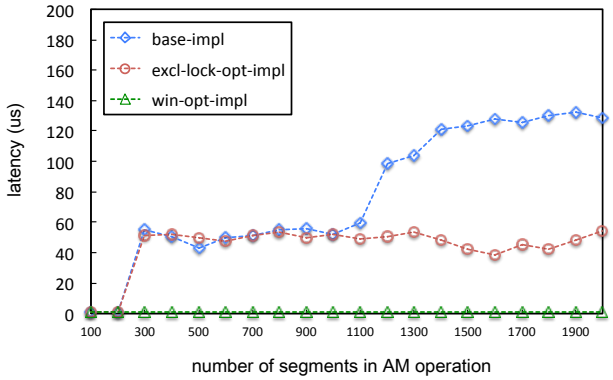
1) Communication Latency: In Figure 5 we measure the latency of a single AM with a *remote search* operation. Two processes are involved in this experiment: the origin process issues one AM operation to the target during the RMA epoch. We vary the message size by increasing the number of segments in the AM. We can see in Figure 5(a) that the *excl-lock-opt-impl* optimization can reduce the latency by around 10% compared with the *base-impl* and that the *win-opt-impl* optimization can further reduce latency by another 10%. We analyze these results by measuring the time spent on synchronization messages in Figure 5(b). The figure indicates that the *win-opt-impl* optimization spends no time on synchronization. This result is expected because with the user buffers already reserved as "exclusive" at window creation time, the origin process does not need to exchange any additional messages in order to reserve user buffers during the RMA epoch. On the other hand, the *excl-lock-impl* optimization does spend some time on synchronization messages, but the time spent does not increase with message size; in comparison, the synchronization time spent by the *base-impl* increases significantly with message size. The reason is that the *excl-lock-impl* optimization needs to send only one synchronization message right after **MPI_WIN_LOCK** in order to reserve all the available user buffers at the target. Since the *base-impl* does not utilize any hints on exclusivity, however, it always needs to wait for previous AM segments to complete execution and reserve new buffers for the rest. Note that both the *base-impl* and *excl-lock-impl* optimizations spend zero time synchronizing when the AM data is less than 200 segments. The reason is that in these cases the AM data is small enough to fit in the system buffer on the target.

Figures 5(c) and 5(d) show a similar experiment but for the *absolute value* operation. We observe that all three implementations perform similarly when the message size is small. As the message size grows, however, the two optimized implementations outperform *base-impl*.

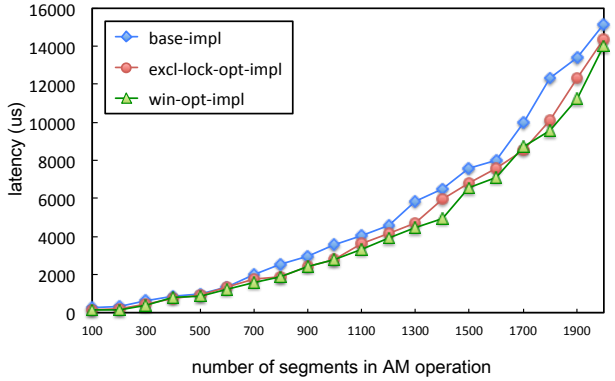
In Figure 5(e), we measure the latency of two AM operations with one **MPI_WIN_FLUSH** in between, with and without any user hint to specify whether user buffers are detached at the target during the **MPI_WIN_FLUSH** operation. As shown in the figure, the **info** hint allows communication latency to improve by around 10%. We further analyze this overhead



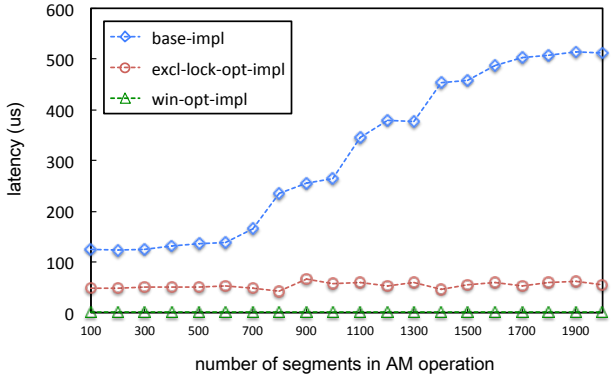
(a) Remote search, latency of one AM operation



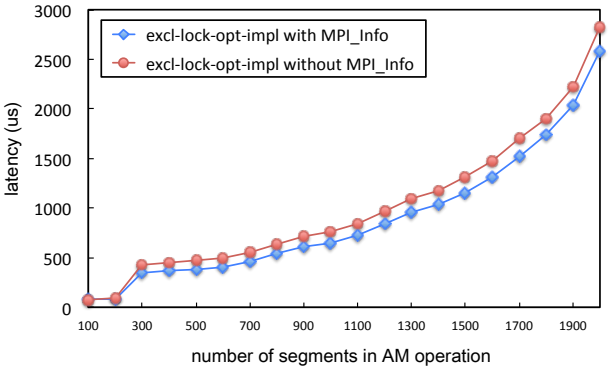
(b) Remote search, latency of synchronization



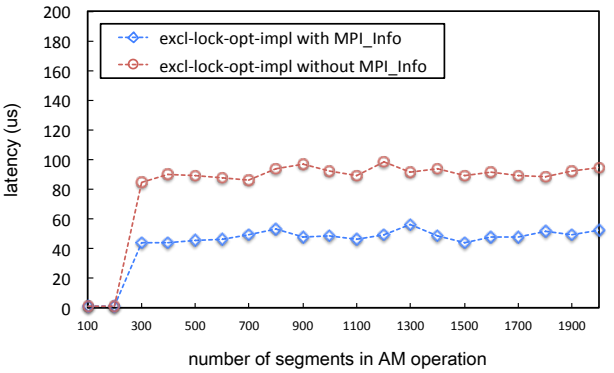
(c) Absolute value, latency of one AM operation



(d) Absolute value, latency of synchronization



(e) Remote search, latency of two AM operations and one flush



(f) Remote search, latency of synchronization

Fig. 5: Communication latency

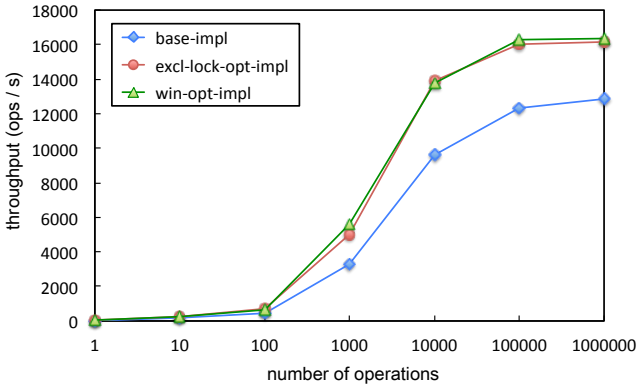


Fig. 6: Operation throughput for *remote search*

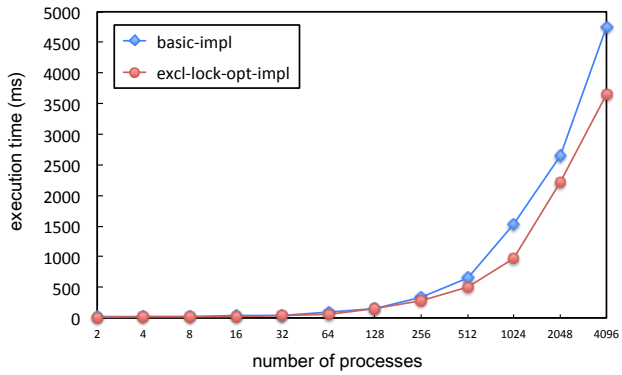


Fig. 7: Scalability performance for *remote search*

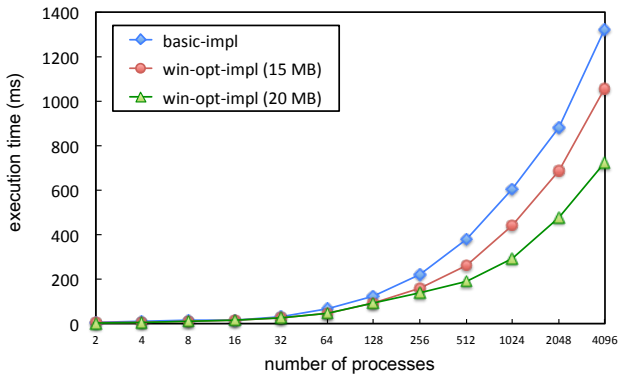


Fig. 8: Contention performance for *remote search*

by measuring just the synchronization time. As shown in Figure 5(f), the synchronization time is consistently reduced by 50%. This result is expected because the **info** hint allows the MPI implementation to have just one handshake operation instead of the two handshake operations when there is no hint from the user. We also studied the *absolute values* benchmark, which has similiar performance but is not presented in this paper because of space restrictions.

2) Operation Throughput: In Figure 6 we measure the AM operation throughput when the origin process issues 1 to 100,000 AMs during the RMA epoch. The figure shows that the *excl-lock-opt-impl* and *win-opt-impl* optimizations can achieve around 25% and 30% improvement over *base-impl*, respectively. The reason is that both the optimized versions have trivial synchronization overhead. On the other hand, in *base-impl*, the origin process must issue at least one synchronization message before each AM and wait for its response before issuing that AM. This approach stalls all following work and limits the peak throughput that is achievable.

3) Scalability Performance: Figure 7 shows the benefit of the *excl-lock-opt-impl* optimization when a large number of origin processes compete for an exclusive lock at the same target. We run the experiment with an increasing number of processes. We observe that at 4,096 processes the *excl-lock-opt-impl* optimization can achieve 20% improvement in performance because of reduced synchronization.

4) Network Contention: Figure 8 shows the benefit of the *win-opt-impl* optimization when we increase network contention. In this experiment, every four origins share a target that is not an immediate neighbor of any origin. Each origin sends a number of AMs to that target. The experiment is set up such that on each origin, all AMs together will consume at most 20 MB of the temporary buffer space at the target. We therefore attach 80 MB of user buffer to the target.

We did two experiments with this framework. In the first experiment, we provide a hint of "20 MB" to each origin. In this case, since all the AMs together can use up at most 20 MB of buffer space, no additional synchronization is needed by the MPI implementation. In the second experiment, we provided a hint of "15 MB" to each origin. In this case, some AMs can be triggered without having to coordinate with the target, but such coordination is required for the remaining AMs. From the evaluation, we observe that even with a hint of "15 MB," the performance improves by 20% at scale. With a hint of "20 MB," the performance improves by 50% at scale.

B. Comparison between MPIX_AM and MPIX_AMV

In our next experiment we measure the throughput of the *remote search* operation using three different AM triggers: (1) *MPIX_AM*, where no packing/unpacking is performed and full output segments are returned; (2) *MPIX_AMV (1.0)*, the vector-based AM trigger where the MPI implementation always performs packing/unpacking and returns packed segments; and (3) *MPIX_AMV (0.8)*, the vector-based AM trigger where the MPI implementation performs packing/unpacking when the percentage of generated data is less than 80%. The maximum sequences count in each output segment is 10.

Figure 9(a) shows the throughput achieved when the amount of generated data increases from 10% to 100% of the maximum output size. When this percentage is below 80%, *MPIX_AMV (0.8)* and *MPIX_AMV (1.0)* perform better than *MPIX_AM* because of the reduced data transmitted. The throughput of vector-based AM keeps decreasing, however, as the percentage of data generated rises. When the percentage is above 80%, their advantage disappears. Further, *MPIX_AMV (1.0)*, which always does packing/unpacking of data, performs around 30% worse than *MPIX_AM* when the generated data approaches 100%. *MPIX_AMV (0.8)*, on the other hand, switches to sending all the data when the generated data is more than 80%, so its performance loss is limited to 10%.

The overhead of the vector-based AM triggers comes from two aspects: (1) runtime packing/unpacking and (2) sending the additional counts array to the origin. To inspect these two aspects, we plot in Figure 9(b) the actual bytes transmitted. The figure shows that at 100%, both *MPIX_AMV (0.8)* and *MPIX_AMV (1.0)* transmit more data than does *MPIX_AM*, the difference coming from the additional counts array that the vector operations need to transmit.

VI. Related Work

Many libraries support AMs, including GASNet [9], IBM's Deep Computing Messaging Framework [10], IBM's Low-level Application Programming Interface (LAPI) [11], and IBM's Parallel Active Message Interface (PAMI) [12]. Unfortunately, these either are low-level machine-specific libraries or are not interoperable with the MPI library. In contrast, the MPI-interoperable generalized AM framework is part of the MPI implementation and achieves both goals.

Previous work has also been done on supporting AMs on top of the MPI library. Examples include AM++ [13] and AMMPI [14]. AM++ is a middle-level library, somewhere between low-level AM libraries such as GASNet and high-level RPC systems such as Charm++ [15] and Java RMI [16]. It has the advantage of allowing message handlers to send arbitrary messages and supports message coalescing and filtering. AMMPI is another implementation of AM over MPI that is used as a compilation target for several PGAS languages. Both AM++ and AMMPI are widely portable to virtually any platform with MPI; however, they are restricted in a number of ways, including lack of asynchronous progress; inability to marshal/demarshal datatypes; and absence of explicitly defined semantics on ordering, concurrency of AMs, and memory consistency. As far as we know, no existing work has been done on supporting AMs within the MPI implementation that can provide such capabilities.

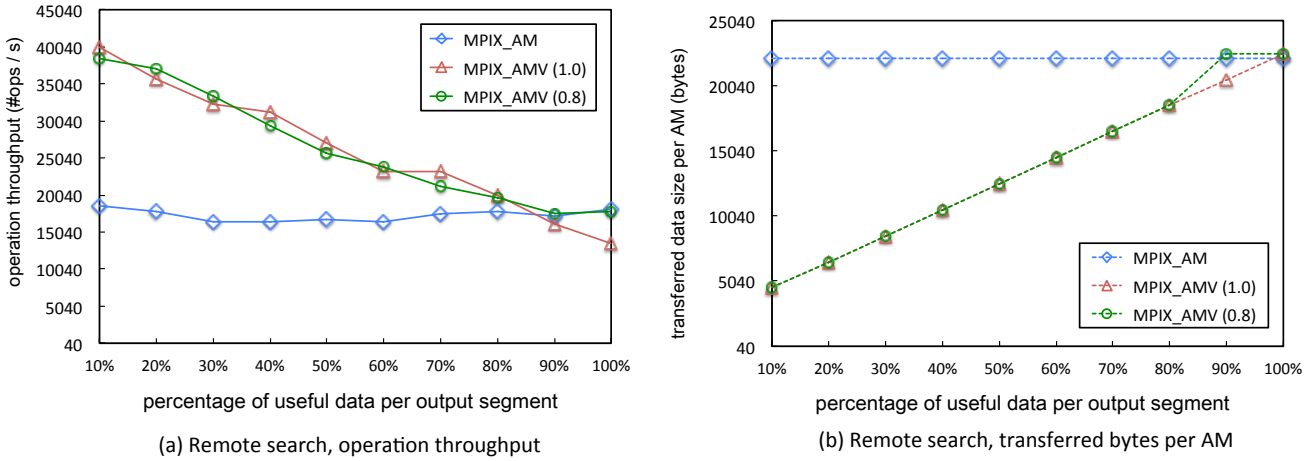


Fig. 9: Operation throughput of **MPIX_AM** and **MPIX_AMV**

VII. Conclusions and Future Work

We analyzed here the performance shortcomings in our previous work on MPI-interoperable AMs [4] and proposed three optimization strategies: reducing redundant synchronization messages automatically or through user hints and improving efficiency of data transmission. We also described a reference implementation of these optimization strategies; and, using a comprehensive set of benchmarks, we demonstrated significant performance improvements in evaluations.

As future work, we plan to investigate the usage of AMs in applications from various domains including bioinformatics (e.g., genome assembly applications) and computational chemistry (e.g., the MADNESS application from Oak Ridge Laboratory, which emulates AM functionality using MPI+threads).

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 *Proceedings of the IEEE International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 1992.

[2] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.0," Sep. 2012, http://www.mpi-forum.org/docs/docs.html.

[3] 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 *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2013.

[4] X. Zhao, P. Balaji, W. Gropp, and R. Thakur, "MPI-Interoperable Generalized Active Messages," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2013.

[5] 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, (under review at the ACM Transactions on Parallel Computing). [Online]. Available: http://www.mcs.anl.gov/~balaji/tmp/mpi3-rma.pdf

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

[7] F. Xia and R. Stevens, "Kiki: Massively Parallel Genome Assembly," https://kbase.us/, 2012.

[8] 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.

[9] D. Bonachea, "GASNet Specification, v1.1," University of California, Berkeley, Tech. Rep. CSD-02-1207, Oct. 2002.

[10] 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 *Proceedings of the International Conference on Supercomputing (ICS)*, New York, NY, USA, 2008, pp. 94–103.

[11] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. J. 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 *Proceedings of the International Parallel Processing Symposium (IPPS)*, Mar. 1998.

[12] S. Kumar, A. R. 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 *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

[13] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New York, NY, USA, 2010, pp. 401–410.

[14] D. Bonachea, "AMMPI: Active Messages—over MPI - Quick Overview," http://www.cs.berkeley.edu/~bonachea/ammpi/.

[15] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993.

[16] J. Waldo, "Remote Procedure Calls and Java Remote Method Invocation," *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, Jul. 1998.