Towards Asynchronous and MPI-Interoperable Active Messages

Xin Zhao,[*] Darius Buntinas,[†] Judicael Zounmevo,[‡] James Dinan,[†] David Goodell,[†]
Pavan Balaji,[†] Rajeev Thakur,[†] Ahmad Afsahi,[‡] William Gropp[*]

[*]University of Illinois, Urbana-Champaign, {xinzhao3,wgropp}@illinois.edu
[†]Argonne National Laboratory, {buntinas, dinan, goodell, balaji, thakur}@mcs.anl.gov
[‡]Queen's University, {judicael.zounmevo,ahmad.afsahi}@queensu.ca

***Abstract*—Many new large-scale applications have emerged recently and become important in areas such as bioinformatics and social networks. These applications are often data-intensive and involve irregular communication patterns and complex operations on remote processes. Active messages have proven effective for parallelizing such nontraditional applications. However, most current active messages frameworks are low-level and system-specific, do not efficiently support asynchronous progress, and are not interoperable with two-sided and collective communications. In this paper, we present the design and implementation of an active messages framework inside MPI to provide portability and programmability, and we explore challenges when asynchronously handling active messages and other messages from the network as well as from shared memory. We test our implementation with a set of comprehensive benchmarks. Evaluation results show that our framework has the advantages of overlapping and interoperability, while introducing only a modest overhead.**

Keywords-**Active messages; MPI; Asynchronous progress; Interoperable; Data-intensive applications**

I. Introduction

In recent years, many new high-performance computing (HPC) applications have become increasingly important in areas such as bioinformatics and social networks. Unlike traditional scientific applications, the computation in these new applications is data-driven, and the topology is always unstructured and dynamically changing throughout the execution. Traditional approaches such as partitioning the work according to the topology and bulk synchronous model no longer suit such applications.

Active messages [1], proposed by Thorsten von Eicken et al. for Split-C in 1992, are a parallel programming paradigm in which 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 receiver executes the message handler to process the data contained in the 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.

Active messages are particularly suited for data-intensive applications and algorithms with irregular communication patterns, such as graph or bioinformatics algorithms. In such algorithms the receiver may not know how many messages to expect or even from which receivers to expect messages. Because active messages do not require the receiver to *post* a receive in order to receive a message, the receiver need not know the communication pattern ahead of time. However, many other algorithms do have regular communication patterns and are well suited to two-sided message passing. Furthermore, an application may have different phases that are better suited to two-sided communication, active messages, or even one-sided communication. Such applications can benefit from a communication library that supports active messages as well as two-sided, one-sided, and collective communications.

Since the application at the receiver does not need to call a function in order for the active message to be processed, the communication library must be prepared to process the message as soon as it arrives. The communication library can be implemented to check for incoming messages only when the application calls a communication function. In this case, if the application does not call a communication function for a period of time, for example if the application is in a long computation loop, then incoming messages will not be processed during that time. Hence, asynchronous processing of messages is important for active messages. Asynchronous message processing can be implemented by using a separate progress thread that checks for incoming messages and processes them. Because this thread is dedicated to receiving and processing messages, the messages are processed immediately, without regard to what the application is doing.

The additional thread can increase overhead, however, because of mutexes needed to synchronize threads when accessing shared data structures. This overhead can affect not only active messages but also two-sided and one-sided messages. The communication library therefore must be designed carefully to minimize the impact of the additional thread.

With modern multicore processors, shared memory can be used to improve the communication performance between processes running on the same node. Many communication libraries make use of shared memory when possible and use network communication for internode communication [2], [3], [4], [5]. An implementation of active messages therefore should use shared memory when possible, in order to improve performance.

In this paper, we chose to add active messages capability to an MPI implementation because MPI is a widely used standard that supports two-sided, one-sided, and collective communications. We modified the MPICH MPI implementation, which is widely portable across many architectures. Our implementation provides asynchronous progress for one-sided and active messages with negligible overhead for two-sided and collective communications. We optimize for the intranode case where the sender can directly access the target buffer of the message.

While one can implement active messages on top of MPI [6], it is difficult to efficiently provide asynchronous progress. The implementation would require a separate thread that would make calls into the MPI library to receive messages and execute the handler. Thus, the MPI library would need to run using the `MPI_THREAD_MULTIPLE` thread level. When an MPI library is running in that level, the library must ensure that every MPI function is thread safe, thus requiring the use of mutexes and imposing an overhead on every communication operation. By implementing active messages inside the library, this overhead can be eliminated for two-sided and collective communications.

II. Overview and Previous Work

We begin with an overview of Active Messages and MPI RMA interface.

A. Active Messages

The active messages paradigm was first introduced in [1] and has since been used internally to implement communications libraries and runtime systems, such as MPI implementations, Co-Array Fortran, and Unified Parallel C. Because existing active messages interfaces were too low-level for applications to directly use them, Willcock et al. designed and implemented AM++ [6], which is a higher-level active messages library intended for applications to use directly.

Low-level active messages interfaces such as GASNet [4], Low-level Application Programming Interface (LAPI) [7], and Parallel Active Message Interface (PAMI) [8] are not suitable for use by applications. While GASNet has been implemented to be portable to different interconnects and architectures, LAPI and PAMI are available only for IBM supercomputers. These interfaces also do not support two-sided or collective communications.

Active messages libraries have been implemented on top of MPI, such as AM++ [6] and AMMPI [9]. Hence, these libraries are widely portable; and applications can use MPI functionality, such as two-sided and collective communications. In order to support asynchronous message processing, however, the active messages library needs an additional thread that waits for incoming messages. Thus, the MPI library must use the `MPI_THREAD_MULTIPLE` thread level, which runs in an *active polling* fashion and always uses the CPU even though no message is coming. Also, it imposes an overhead due to thread synchronization and mutexes on every communication operation.

We classify the MPI-based active messages into three categories:

- NO-ASYNC – Asynchronous message processing is not supported.
- THREAD-ASYNC – Asynchronous message processing is provided by using a thread above the MPI library.
- INTEGRATED-ASYNC – Asynchronous message processing is provided internally by the MPI implementation.

AMMPI and AM++ belong to the NO-ASYNC class. However, if the application using AMMPI or AM++ creates a thread to wait for incoming messages, that usage would fall into the THREAD-ASYNC class. In this paper, we propose a strategy that falls into the third class: INTEGRATED-ASYNC. It can support asynchronous message processing by an internal thread and handles active messages in a more efficient way.

B. MPI RMA Interface

The first MPI standard provided only two-sided and collective communications. The MPI-2 standard, which was released in 1997, added functionality for one-sided communication (also called remote memory access (RMA)). One-sided communication allows one process, the *origin* process, to specify all communication parameters, for both the origin and the destination, or *target*, process. The active messages API in our work is based on the MPI RMA interface.

MPI-2 defines three types of communication operations: `Put`, `Get`, and `Accumulate`. `Put` and `Get` operations transfer the data to and from a window on the target process. The `Accumulate` operation combines the data from the origin process with the data on the target process using a predefined operation.

In order to make sure that all one-sided operations are finished, explicit synchronization modes are defined and used, including *active target mode* and *passive target mode*. Active target mode has two synchronization mechanisms: FENCE (Fig.1(a)) and POST-START-COMPLETE-WAIT (PSCW) (Fig.1(b)). Passive target mode has one synchronization mechanism: LOCK-UNLOCK (Fig.1(c)). For detailed semantics of synchronizations in the MPI RMA interface, please refer to [10].

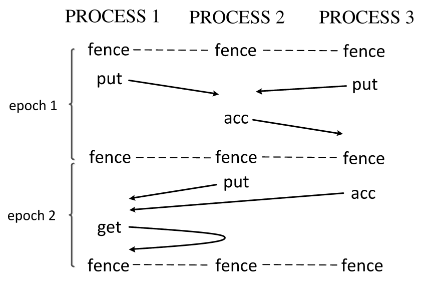
III. Design Issues

In this section, we describe how to extent MPI accumulate operation to support user-defined function and message pipelining technique.

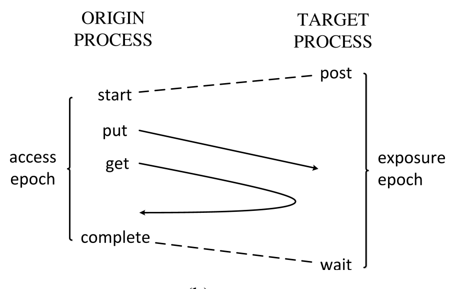
A. User-Defined Function and Operation Registration

The `Accumulate` operation as defined in the MPI standard is similar to the concept of active messages except for the predefined operations. `Accumulate` allows users to specify simple calculations to be performed in remote memory. However, it supports only a limited set of built-in operations and does not support user-defined operations. To allow more complex operations on remote processes, we extend the functionality of `Accumulate` to support user-defined functions.

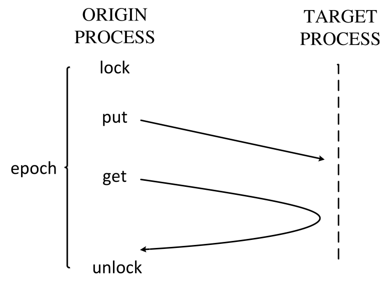
In the MPI standard, user-defined functions are allowed for `Reduce` operations. We use the same mechanism to specify user-defined functions for `Accumulate`.



(a) FENCE



(b) PSCW



(c) LOCK-UNLOCK

Fig. 1: Synchronization types in MPI RMA interface

User-defined functions have the following prototype: `void MPI_User_Function (void* invec, void* inoutvec, int* len, MPI_Datatype* datatype)`. The `invec` and `inoutvec` parameters describe arrays that the function combines, while the `len` and `datatype` parameters describe the layout of the arrays. Each invocation of the function leads to the pairwise execution of user-defined operation on array elements. The result of the function is written to `inoutvec`. After the function has been defined, the `MPI_Op_create` routine should be called to bind the function to an operation *handle* that can be subsequently passed to the `Accumulate` function.

We also propose two new routines for operation registration: `MPIX_Op_register(MPI_Op op, int id, MPI_Win win)` and `MPIX_Op_deregister(MPI_Op op, int id, MPI_Win win)`. Each of them is a collective call among all processes that are in the same window. When `MPIX_Op_register` is called, the operation handle and a *handler id* are passed and internally stored in a hash table on that window. Because operation handles are local, the *handler id* is used to identify handler functions on remote processes. In this way, the user-defined operation can be "available everywhere." Operations defined on different processes with the same handler id must have equivalent functionalities.

B. Message Pipelining

The MPI standard specifies that the implementation may call a user-defined function multiple times in order to handle a single large message. If the source and target buffers are arrays, then the user-defined function may be called to handle smaller portions of the array. This approach allows the message reception to be pipelined with the execution of the user-defined function. It also reduces the memory required in order to buffer the incoming message. We implemented pipelining by receiving a *chunk* of the incoming data, then calling the user-defined function specifying the data received and the corresponding data in the target buffer.

IV. Asynchronous Progress Engine

In most MPI implementations, the MPI library typically calls the progress engine to handle incoming messages only when an MPI routine is called. In order to improve performance for handling intranode messages using shared memory, most MPI implementations use active polling and do not block while waiting for a message. Although this approach improves intranode communication performance, it has the effect of using the CPU even when no message is being handled. Some MPI implementations do provide asynchronous progress and non-busy-waiting [11], but these features come with a performance penalty.

MPI implementations typically have a single progress engine that handles both one-sided and two-sided messages. Having a single progress engine has two disadvantages: (1) one-sided and active messages cannot be processed immediately upon arrival but have to wait until the target explicitly calls an MPI routine; and (2) one-sided and active messages vs. two-sided and collective messages cannot be handled in parallel but have to be processed in serial through the single progress engine.

Our implementation has two progress engines: one new asynchronous progress engine handling active messages and one-sided messages, and the regular progress engine handling two-sided messages and collective messages. Active messages and one-sided messages can be processed upon arrival by this separate progress engine and can be processed concurrently with other kinds of messages. The following subsections describe important issues in designing and implementing the asynchronous progress engine for both network and shared memory.

A. Network Solution

We use a separate internal thread in the network module to wait for active messages and one-sided messages from the network. When an MPI process encounters the first window creation routine (indicating that there will be active messages or one-sided messages coming), it internally spawns a separate thread used by the asynchronous progress engine. This thread is terminated when `MPI_Finalize` is called. It does not wait for messages from shared memory, and therefore it can block while waiting for incoming messages with minimal impact on performance. The original progress engine is still used by the main thread to handle two-sided and collective messages. Currently we have implemented the separate thread in the TCP network module.

B. Shared-Memory Solution

Because the separate thread described above calls the asynchronous progress engine in a blocking way, it is not applicable for messages from shared-memory communication. We solve

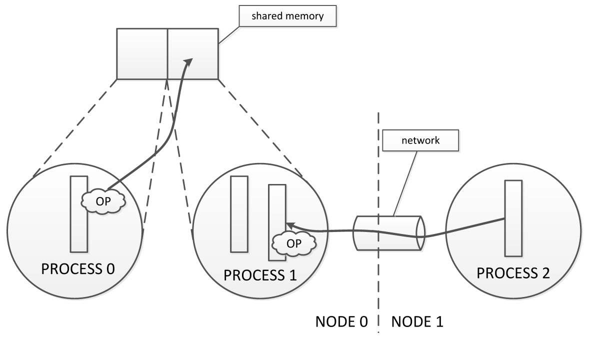


Fig. 2: Working scenario of asynchronous progress engine

this by enabling "origin computation." When the origin process encounters an active message or one-sided message targeting at a remote process on the same node, it directly fetches the remote data, does the computation (`AM` or `Accumulate`) locally, and pushes it back to the remote process. By enabling "origin computation," active messages and one-sided operations can be handled asynchronously without the help of a separate thread.

To support the direct access to memory of a remote process on the same mode, during window creation phase each MPI process allocates a shared-memory region and exchanges the shared-memory address information. The MPI standard has two methods for creating an MPI RMA window. With `MPI_Alloc_mem` + `MPI_Win_create`, the first routine is used to allocate a memory region and returns the memory address; the second routine is used to create an MPI RMA window by using a memory address in the argument list. With `MPI_Win_allocate`, a new routine in the MPI-3 standard, memory allocation and window creation are performed together in one single function call. We implemented the shared-memory solution in both methods.

The user can enable or disable the asynchronous progress engine by passing an `Info` argument to window creation routines. If the asynchronous progress engine is enabled, the MPI process internally allocates a shared-memory region for the window; if there are processes on different nodes, it internally spawns a separate thread. Figure 2 demonstrates an example of how the asynchronous progress engine works. In the example, process 0 and process 2 are going to issue active messages to process 1. Process 0 is on the same node with process 1, whereas process 2 is on a different node. Process 1 has a separate internal thread to handle active messages from process 2, while process 0 performs "origin computation" and directly writes results to the shared-memory region of process 1. All active messages operations are performed asynchronously.

Because lock and unlock requests are handled by the original progress engine and hence are handled only when the main thread calls an MPI function, when supporting origin computation on shared memory, the origin process must be able to acquire the passive lock on the target process without explicitly sending out a lock request message. We solved this problem by implementing a distributed lock [12] in the shared-memory region used by shared memory and network processes.

C. *Thread Safety and Process Safety*

For the network solution, key data structures that are shared between two progress engines (including sockets and file descriptor tables, sending queues, and pending receiving packets) are either duplicated or protected by mutexes in order to avoid data races. In active target mode, a counter is used to determine whether all operations have arrived and whether ending synchronization can return. It is atomically accessed by two threads. In passive target mode, both the main thread and the asynchronous thread on the target process may try to acquire the passive lock. We added a mutex on the passive lock to avoid data race.

For the shared-memory solution, because the origin process performs origin computation and then writes results into the memory of the target, both the origin and the target processes may write to the same memory address concurrently. We added interprocess mutexes in the shared-memory region of every process to avoid this case. Those mutexes are disabled by default, but the user can enable them by passing an assert to synchronization calls indicating a possibility of conflicting writes happening in the memory of the target process.

V. Experimental Evaluation

In this section, we present the performance results of AM-MPI on the Fusion cluster at Argonne National Laboratory. Fusion has 320 nodes, each having two Intel Xeon X5550 quad-core CPUs, and QDR InfiniBand HCAs. We implemented the AM-MPI framework based on MPICH2-1.4.

We present latency, overlapping, two-sided/AM interoperability and stencil communication tests. We tested the `AM`, `Put`, and `Accumulate` on three different IPC (interprocess communication) structures: network-only, shared-memory-only, and network-and-shared-memory. All tests are compared under the following three execution models:

1) EXT-ASYNC:AM-MPI with external asynchronous thread
2) INT-ASYNC:AM-MPI with internal asynchronous thread
3) NON-ASYNC:AM-MPI without asynchronous thread

EXT-ASYNC is enabled by passing an `Info` argument to the window creation routine; INT-ASYNC is enabled by turning on the environment variable `MPICH_ASYNC_PROGRESS` in MPICH, which would spawn a separate thread to actively poll the progress engine.

For Sec. V-A, V-B, V-D, all results are gathered with FENCE; for Sec. V-C, all results are gathered with FENCE (active) and exclusive LOCK-UNLOCK (passive). Similar results are observed for other synchronizations, but we cannot show them here because of space limitations. For figures in the following subsections, IB refers to InfiniBand, and PTP refers to point-to-point communication.

A. *Overhead of Active Messages Handler*

Fig. 3 shows the overhead of the AM handler by comparing it with `Put` and `Accumulate` operations on network and

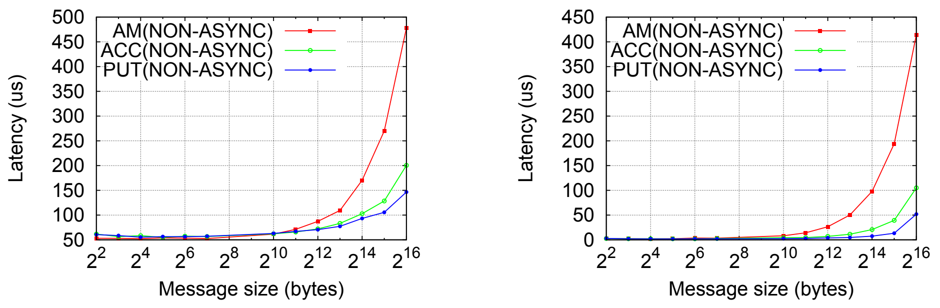


Fig. 3: Overhead of AM handler

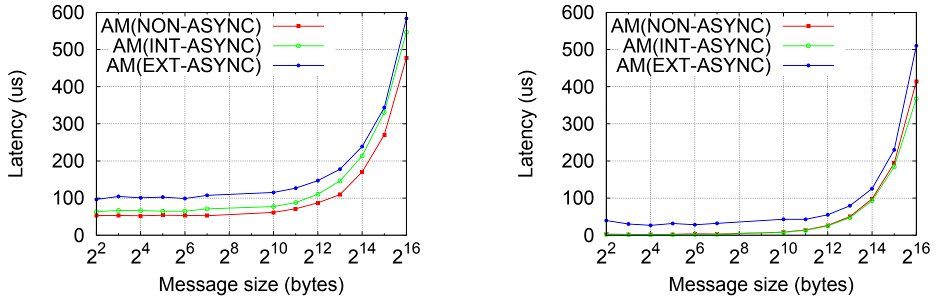


Fig. 4: Latency of single AM operation

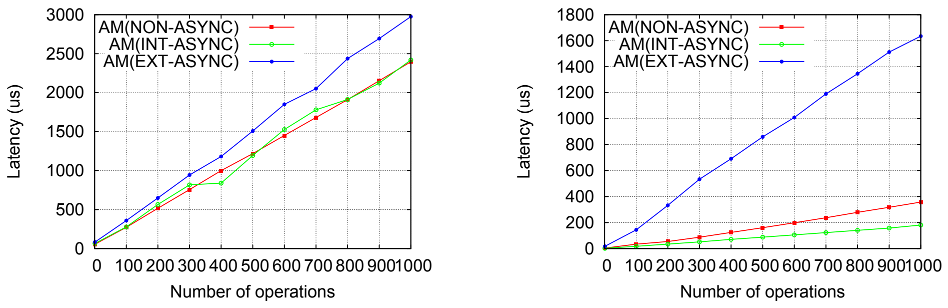


Fig. 5: Latency of multiple AM operations

shared memory under the NON-ASYNC model. In this test, one message is transmitted from the origin process to the target process, and latency is measured with increasing message sizes.

For `Put` operations, there is only data copy on the target process; therefore, it has the smallest overhead. For `Accumulate` operations, a predefined function is called upon data arrival, which has an optimized implementation in MPICH. For `AM` operations, a user-defined function is called when data is received; and it has the highest overhead. Here we present only the overhead with FENCE. Similar results are observed for PSCW and LOCK-UNLOCKs, but we cannot show them here because of space limitations.

B. Communication Latency

Figure 4 shows the comparison of communication latency for a single `AM` operation among the various IPC models for varying message sizes. Figure 5 shows the latency for multiple `AM` operations through various IPC models for a fixed (4-byte) message size. While INT-ASYNC and NON-ASYNC are mostly on par, EXT-ASYNC underperforms. Actually, when `MPICH_ASYNC_PROGRESS` is enabled, MPICH operates at the `MPI_THREAD_MULTIPLE` level, making the progress engine a critical section that the main application thread and the EXT-ASYNC thread must compete for. In comparison, INT-ASYNC operates a separate progress engine that is not shared with the main thread. The overhead of the EXT-ASYNC thread entering and exiting the mutex-protected progress engine becomes apparent when the number of operations grows (Fig. 5). Additionally, for shared-memory communications, INT-ASYNC performs better than NON-ASYNC when the number of operations increases (Fig. 5(a)), because it does directly memory copy instead of copying the messages through the Nemesis [2] send queues.

C. Overlapping Effects

To measure the overlapping effects of the asynchronous progress engine, we crafted tests in which a certain amount of computation is performed on the target process while the origin process sends multiple active messages. The target process issues no MPI call while it computes. In Fig. 6, we show the overlapping effects for the passive target mode.

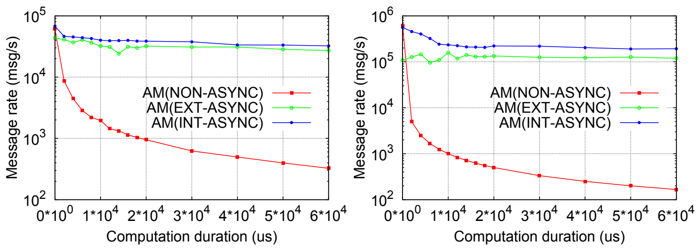


Fig. 6: Overlapping effects

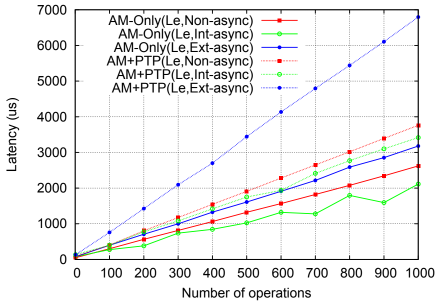
In Fig. 6(a), both EXT-ASYNC and INT-ASYNC have overlapping effects, whereas NON-ASYNC has none. The reason is that while the main thread is busy doing computation, EXT-ASYNC and INT-ASYNC can separately handle communications with the progress thread whereas NON-ASYNC cannot. Figure 6(b) shows the overlapping effect of the shared memory implementation. In the EXT-ASYNC model, the asynchronous thread on the target process deals with incoming active messages while the main thread is doing computation; in the INT-ASYNC model, after the origin process acquires the passive lock on the target, it directly performs the computation and writes the results into the memory of the target while the target is doing computations. Here there is a performance gap between EXT-ASYNC and INT-ASYNC on both network and shared memory, due to the queuing operations needed for distributed passive lock.

No overlapping effects are observed for the active target mode. Because AM-MPI is implemented based on the RMA implementation in MPICH, in which all messages are delayed and lazily issued out during the ending synchronization phase, message reception on the target must also happen within the ending synchronization phase, which cannot be overlapped with computation in the epoch. Future implementations of MPICH are expected to have support for eager issuing of messages; in that case overlapping effects can be expected for active target mode.

D. Interoperability Performance

Figure 7 demonstrates the execution time when active messages and two-sided communications happen together (AM+PTP). In the test, the origin process sends multiple active messages to the target process. At the same time, a third process sends multiple two-sided messages to the same target process. We increase the number of both AM and two-sided operations, measure the execution time on the origin process of AM communications, and compare the results with Fig. 5 (AM-Only).

Figure 7(a) shows the interoperability performance on IB network. When the two-sided communication is added, the time of both the EXT-ASYNC and NON-ASYNC models increases, because under EXT-ASYNC and NON-ASYNC, active messages and two-sided messages are processed by the same thread (asynchronous thread under EXT-ASYNC and main thread under NON-ASYNC). However, the time of INT-ASYNC does not increase obviously. The reason is that under INT-ASYNC, active messages and two-sided messages are processed in separate threads: the asynchronous thread handles active messages, and the main thread handles two-sided messages. Therefore, adding two-sided communications does not affect the total execution time in INT-ASYNC.



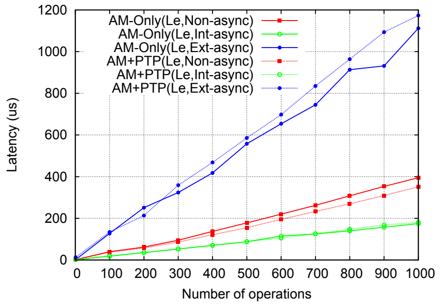


Fig. 7: Interoperability performance

Figure 7(b) shows the interoperability performance on shared memory. When adding two-sided communications, the execution times in all three models are increased, because now active messages and two-sided messages are handled within the same thread for all three models. However, since there is no extra thread existing or awake in NON-ASYNC and INT-ASYNC, the overhead is relatively small compared with that of EXT-ASYNC.

E. Stencil Kernel Benchmark

To examine the scalability and the effectiveness of the asynchronous progress engine on both network and shared memory, we implemented a stencil kernel benchmark using AM operations and FENCE synchronization. During runtime, every node has 8 MPI processes. For the INT-ASYNC and EXT-ASYNC models, each MPI process would spawn a separate thread. Processes are formed as a square grid, and each one sends and receives messages from neighbors within distance 1. The process number is scaled from 2×2, 4×4 ... to 20×20.

Fig. 8 demonstrates the execution time with small grid size and large grid size. Both INT-ASYNC and EXT-ASYNC performs worse than NON-ASYNC due to overhead of the context switching. INT-ASYNC has less overhead than EXT-ASYNC as expected, because asynchronous thread in INT-ASYNC is waken up only when there are active messages on network coming, whereas asynchrnous thread in EXT-ASYNC is always working throughout the whole execution and context switching is very frequent.

F. Graph 500 Benchmark

Graph 500 [13] is a relatively new supercomputing benchmark used to test data-intensive computing systems. It performs a breadth-first search, and its performance metric is traversed edges per second (TEPS). The MPI one-sided implementation of Graph 500 performs a large number of 8-byte `MPI_Accumulate`s among all the peers during FENCE epochs. A straightforward improvement upon that approach is to coalesce a certain number of those `MPI_Accumulates`, in order to reduce the large number of small communications with each peer. Such an improvement can simply use MPI derived datatypes (DDTs) to send coalesced data resulting from the local accumulation of the data meant for each peer in each fence epoch. However, the DDT approach must satisfy the nonoverlapping constraint imposed by the MPI specification for target datatypes in `MPI_Accumulate`. By resorting to AM, which uses user-defined functions instead of a regular `MPI_Op` in `MPI_Accumulate`, the aforementioned constraint can be totally avoided. Before issuing operations to each peer at the end of the epoch, the AM approach compacts the locally accumulated data into sparse arrays. This step, which occurs once for each peer in each epoch, is less computation-intensive than the nonoverlapping constraint checking that the DDT approach performs for each local accumulation.

In Fig. 9, we present measurement of TEPS to compare the default one-sided implementation (Default-g500) with a DDT-based implementation (DDT-g500) and our AM-based implementation (AM-g500). All results are gathered under NON-ASYNC model. The tests are performed for 2^{15} vertices (scale-15) and 2^{20} vertices (scale-20) respectively over 128, 256, and 512 processes.

For scale-15 (Fig. 9(a)), both DDT-g500 and AM-g500 perform better than Default-g500; and as expected, AM-g500 performs even better than DDT-g500. The same trend is observed in scale-20 (Fig. 9(b)) over 128 and 256 processes. However, AM-g500 performs poorly over 512 processes at scale-20. The case where AM-g500 underperforms compared with DTT-g500 and Default-g500 is a consequence of a few more general behaviors of the one-sided implementation of Graph 500. For DDT-g500 and AM-g500, job size variations create a tradeoff between the size of communication with each peer and the number of peers each process communicates with. The larger the job, the larger the number of peers but the smaller the message sizes. As a result, the coalescing effect tends to be watered down when the job size increases at a fixed scale. A point can be reached where the overhead of each coalescing approach will not be offset enough by the communication time saved by coalescing. The important observation, however, is that at a fixed scale, Graph 500 remains scalable with respect to the number of processes until a peak is reached, and then TEPS enter a phase of performance drop. The numbers of processes shown in Fig. 9 happen to be in the range of performance drop for scale-15 and scale-20. We confirm in particular that 128 is the optimal job size for scale-20 for our test environment when job sizes are changing by a factor of 2. Consequently, no matter what implementation chosen, the informed user is less likely to execute Graph 500 over job sizes where the AM implementation underperforms.

VI. Discussion and Future Work

The existing prototype of MPI user-defined function discussed in Sec. III has several restrictions:
(1) Only one kind of datatype layout and size is accepted by the user function, which forces input and output data to have the same data layouts.
(2) `MPI_Accumulate` allows updates only on the data specified by the function call, which makes some operations (inserting an element into a remote queue) hard to implement by using `AM` operations.
(3) For noncontiguous datatypes data to be transferred between two processes, it must be internally packed at the origin side, and unpacked at the target side; therefore the target process has to allocate a large temporary buffer to place the unpacked data. This approach is inefficient when encountering sparse data.

To solve these problems, we propose a design of generalized API for active messages in MPI. This API allows for much more flexibility on usage and allows the user to specify the type of receiving buffer by themselves. The generalized API includes following new functions: `MPIX_Am_handler_register`, `MPIX_Am_free`, `MPIX_Am_send`, `MPIX_Am_hhandler` (header handler), and `MPIX_Am_comp_handler` (completion handler). Only the header handler needs to be registered. When `MPIX_Am_send` is called, the runtime system issues small immediate data followed by the main data. Immediate data is handled by the header handler at the target. The header handler specifies the receiving buffer and the completion handler for the main data. When the main data arrives, all AM operations are done in the completion handler. Because users can specify the datatype of the receiving buffer, they can make it as a packed datatype, in order to avoid memory efficiency issues. Also, the argument of the completion handler is a pointer to a user-defined C structure; therefore users can pack any arguments they need and pass these arguments to the function, which removes restrictions on input data. We plan to implement this generalized API in our future work.

VII. Conclusion

We have presented the design and implementation of active messages in MPI based on the MPI RMA interface. This implementation is achieved by proposing operation registration schemes, extending the functionality of the `Accumulate` operation to support user-defined message handlers, and using multithreading and shared-memory allocation to support asynchronous progress engine. The impact of this work is as follows: active messages can work interoperably with two-sided and collective communications while introducing a modest overhead; a process that does shared-memory communication directly performs origin computation that allows the

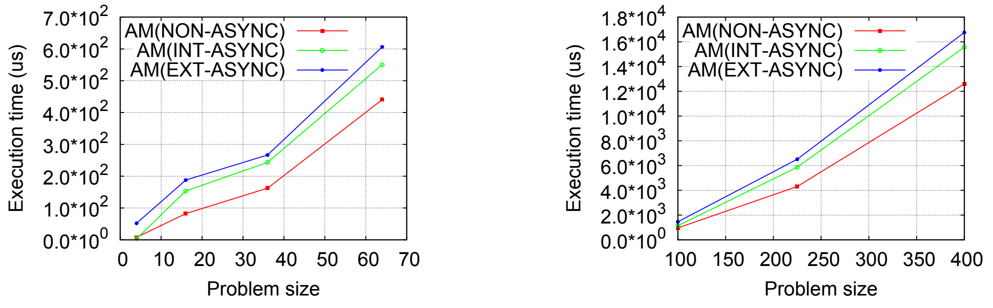


Fig. 8: Execution time of stencil kernel benchmark

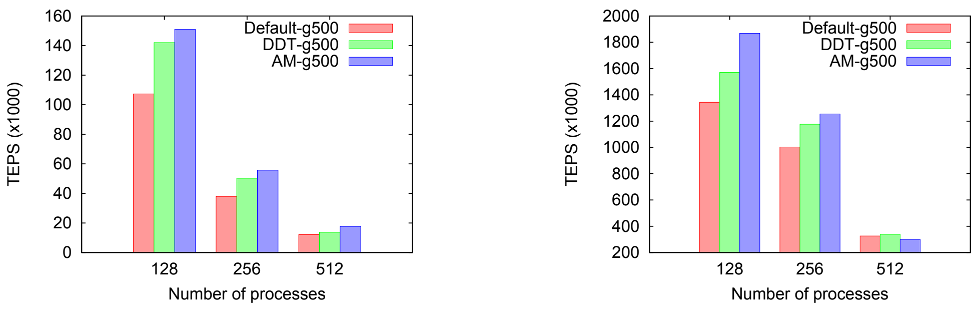


Fig. 9: Graph 500 comparative performance results

asynchronous thread to nonactively wait for messages from the network; and asynchronous AM is implemented inside MPI, not on top of MPI. Through the evaluation of communication latency, overlapping effect, interoperability, and stencil tests, we demonstrated that the performance is competitive with external asynchronous progress engine. Furthermore, we investigated the performance of the Graph 500 benchmark by comparing an AM with a derived datatype implementation, demonstrating the advantage of active messages in implementing data-intensive algorithms.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We thank Ziaul Haque Olive who provided the derived datatype implementation of Graph500 benchmark. We gratefully acknowledge the computing resources provided on "Fusion", a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

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 19th annual international symposium on Computer architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266.

[2] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, S. J. Turner, B. S. Lee, and W. Cai, Eds. Washington, DC, USA: IEEE Computer Society, May 2006, pp. 521–530.

[3] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

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

[5] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[6] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 401–410.

[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 *Proceedings of the International Parallel Processing Symposium*, March 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 26th International Parallel Distributed Processing Symposium (IPDPS)*, may 2012, pp. 763–773.

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

[10] Argonne National Laboratory, "MPICH," http://www.mpich.org/.

[11] IBM, "Cluster product libraries, parallel enviroment runtime edtion," http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp/.

[12] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, pp. 21–65, 1991.

[13] Graph500, "Graph500," http://www.graph500.org/.