Runtime Support for Irregular Computation in MPI-Based Applications

Xin Zhao,* Pavan Balaji,† (Co-advisor) and William Gropp* (Advisor)

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

***Abstract*—In recent years more and more applications have been using irregular computation models in various domains such as bioinformatics and social network analysis. Traditional data movement approaches are not well suited for such applications because of the irregular communication patterns, sparse data structures, fast growth rate of data movement as system size or problem size rises, and so forth. Active Messages (AM) is an alternative programming paradigm that is more suitable for irregular computations. It allows small pieces of data to be dynamically moved to the remote process and certain computation to be triggered, and the remote process does not need to explicitly receive the data. In this paper, an outline of the first author's Ph.D. thesis, focusing on runtime support for irregular computation, is presented. In the first part, we combine the capability of AM with traditional MPI data movement patterns; and we propose a generalized MPI-interoperable AM framework (MPI-AM). In the second part, we extend the MPI-AM framework to provide a model of dynamic task parallelism for data-driven computation. In each part we describe critical issues, demonstrate the current status of the work and performance gain, and discuss remaining challenges to be solved.**

I. Introduction

Irregular computation models have become increasingly important in recent years. These models are widely used in applications in various domains such as bioinformatics (SWAP [1] and Kiki [2]), computational chemistry (MADNESS [3][4]), and graph algorithms in social network analysis. Irregular models differ significantly from traditional models: (1) some of them are organized around sparse structures, (2) the data movement pattern is often irregular and data dependent, and (3) the growth rate of data movement as the systems size or problem size increases is much faster than that of computation. Unfortunately, the traditional SEND/RECV- and PUT/GET-like data movement approaches are not well suited for such irregular computations.

The Active Messages (AM) model [5] is an alternative parallel programming paradigm that is more suitable for irregular computations. It allows the sender to move a small piece of data to the receiver and to trigger computation; the receiver does not need to explicitly receive the data. Since the Message Passing Interface (MPI) [6] is the de facto standard for parallel programming on large-scale systems, a programming model that can combine the traditional MPI model and the AM-based model is needed. This is the first part of the first author's Ph.D. thesis: we propose a generalized framework, called MPI-interoperable Active Messages (MPI-AM), to support both irregular and regular capabilities in MPI runtime. The framework allows an application to be modified incrementally in order to use AM only when necessary, avoiding rewriting the entire application.

Since one process can use AM to bring small pieces of data to the remote process and to trigger computation remotely via an AM handler, each handler can be treated as a "task," which is a combination of computation and data. One question of interest arises: Can MPI-AM be used to build a dynamic tasking framework to support data-driven computation? This is the focus in the second part of the thesis. In order to achieve this goal, extending the MPI-AM framework by enabling MPI calls from the AM handler is essential. Doing so involves addressing issues such as locality: since the process of executing the computation and the process of storing the data are not necessarily the same, one must consider the impact of different AM data locality choices on both correctness and performance. Other issues that we would like to investigate include thread safety, dependent execution of tasks, and load balance among tasks.

In summary, the thesis focuses on providing efficient and scalable support from the MPI runtime, to support Active Messages and the tasking model, for irregular computation. It includes two subtopics:

- Integrated data and computation management
  - Generalized MPI-interoperable AMs (MPI-AM)
  - Asynchronous processing on MPI-AM
  - Efficiency of MPI-AM with irregular applications
- Dynamic task parallelism for data-driven computation
  - Enabling of MPI calls within the AM handler
  - Dependent execution of tasks
  - Impact of data locality choices and load balance
  - Efficiency of tasking framework with applications

Currently the first two parts in the first subtopic have been finished ([7][8][9]), and work on the third part has begun. The second subtopic will be addressed next.

II. Integrated Data and Computation Management

In the first part of the thesis, we propose a generalized framework for MPI-interoperable AMs (MPI-AM). Previous work on supporting AM on top of MPI includes AM++ [10] and AMMPI [11]. While portable, they lack explicit semantics about ordering, concurrency of AMs, and memory consistency; and their implementations lack asynchronous progress.

The workflow of the MPI-AM framework is illustrated in Figure 1. We propose a new routine, **MPIX_AM**, for issuing AMs and a new prototype, **MPIX_AM_USER_FUNCTION**,

for specifying the AM handler. With these routines, users can manage data content and movement among five associated buffers: origin input buffer, target input buffer, target persistent buffer, target output buffer, and origin output buffer. Note that the target input buffer and target output buffer are internal buffers associated with each AM handler whereas the rest are public buffers. As shown in Figure 1, when **MPIX_AM** is called, the origin input data is sent to the target and is staged in the target input buffer. This staged data serves as the input to the AM handler, and the handler stores its output into the target output buffer. Once the computation in the handler is completed, the output data is sent back to the origin output buffer. The target persistent buffer stores data that already exists at the target's window and is accessed within the AM handler. All updates on this buffer can be seen by future operations. In the following subsections we examine critical issues raised by MPI-AM.

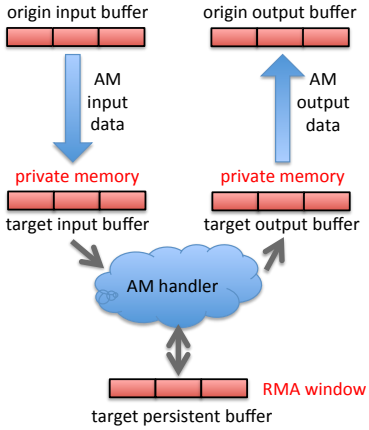

Figure 1: MPI-AM workflow

A. Performance Issues and Correctness Semantics

Among the major issues are those involving message streaming and buffering and correctness semantics.

1) Data Streaming and Buffer Management: For a large AM, we propose the concept of *segments*. The purpose is to allow the MPI runtime system to choose how to split it into smaller units in order to achieve a pipeline effect and reduce space for staging data. *Segments* are used to let the user specify the minimum granularity of splitting AMs in order to trigger the corresponding handler. For example, in the DNA assembly application [1][2], the handler needs at least one DNA sequence in order to perform the computation. Therefore the user needs to specify *segment* as one DNA sequence.

Since each AM handler is associated with the target input buffer and target output buffer, a critical question is who is responsible for managing such temporary buffers. While the MPI implementation might choose to internally allocate temporary buffers, that is only a runtime optimization; and the user cannot assume the availability of such buffers. The user's responsibility is to provide enough AM temporary buffers to the MPI implementation. We propose routines to attach/detach buffer for this purpose. We note that the size of the user-provided buffer must be large enough to accommodate at least one AM segment. Also, since the user buffer is shared among AMs from all origins, the origin process needs to perform synchronization internally with the target in order to reserve a portion of buffer before it can issue the AM data.

2) Correctness Semantics: Here we discuss critical issues in correctness semantics, including ordering, concurrency, and memory consistency. Because of space limitations, considerations such as atomicity or thread safety are not presented here; please refer to [7] for details.

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, for AMs 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 when multiple AMs access it. Furthermore, we allow the user to release the ordering of AMs by passing MPI information during window creation. Reduced ordering can be beneficial for some applications, for example those that use AMs only to read the target data but do not update it.

Concurrency: When one or more origins issue multiple AMs to the same target, the target can process them either simultaneously or in serial. While concurrent execution has an obvious potential benefit for performance, it requires careful use with respect to its data accesses and must avoid conflicts with other AMs by using atomic operations or locks. 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 situations where concurrency is inconsequential.

Memory Consistency: MPI RMA defines two memory models: UNIFIED (one single copy of window) and SEPARATE (one *public* window copy and one *private* window copy) [6]. Because of limited space, here we will not explain window memory semantics in detail. We recommend that the reader consult related books and papers ([6][12][13]) to better understand the semantics. One main difference between AM and RMA operations is that RMA operations access the *public* window whereas AM handlers access the *private* window. The reason is that operations involved in the AM handler are essentially local loads/stores invoked by the target process and are not like puts/gets/accumulates invoked by the origin process. This difference raises several subtle interoperability issues. For example, in the SEPARATE model, if an AM and a regular RMA operation concurrently update nonoverlapping memory regions in the same window, the state of the data in that window is undefined. 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 the same cache line, such an update would be overwritten when the cache line is written back to memory. The same issue also exists for two concurrent AMs. Further, in both memory models, each AM handler must ensure that it sees updates by previous operations and leaves the window in a consistent state for future operations. To alleviate this situation, in the SEPARATE model, the MPI implementation needs to flush the cache back to memory before returning the AM completion notification to the origin. In both the SEPARATE and UNIFIED models, the MPI implementation needs to perform a full memory barrier before invoking the AM handler and after finishing the AM handler, in order to ensure the ordering between operations before the handler and within the handler, and the ordering between operations within the handler and after the handler.

B. Asynchronous Processing on AMs

Having a single progress engine to handle all incoming messages has the disadvantage that the runtime has to delay processing AM/RMA messages until the process explicitly makes an MPI call. Having an asynchronous progress engine for AM/RMA can significantly improve the performance by processing messages immediately upon arrival and concurrently with other MPI messages. In our previous work [9], we presented an implementation that can provide asynchronous processing on AMs internally from the MPI runtime. To avoid active polling, we use *origin computation* to asynchronously handle AMs from shared memory. The runtime first allocates a shared-memory region among processes on the same node during window creation; when one process issues an AM targeting at another process on the same node, it directly fetches the data from the target process's shared-memory region, performs the computation locally, and writes results directly back into the target process's memory. For internode communication, a separate internal thread is generated in the network module waiting for AMs coming from the network. The thread can minimize impact on performance by blocking during waiting time, because it does not need to handle intranode messages.

C. Performance Evaluation

Figures 2(a) and 2(b) respectively show latency and throughput achieved for different numbers of segments per AM packet. All microbenchmark tests are conducted on a 310-node system in which each node has 16 cores. The nodes are connected with QLogic QDR InfiniBand. The AM latency reaches its lowest at a pipeline unit size of 40 segments, where it achieves a perfect balance of computation and data movement.

We use the Graph500 benchmark [14] for kernel benchmark testing. All tests are run on a 320-node (8 cores per node) system connected with Mellanox QDR InfiniBand.

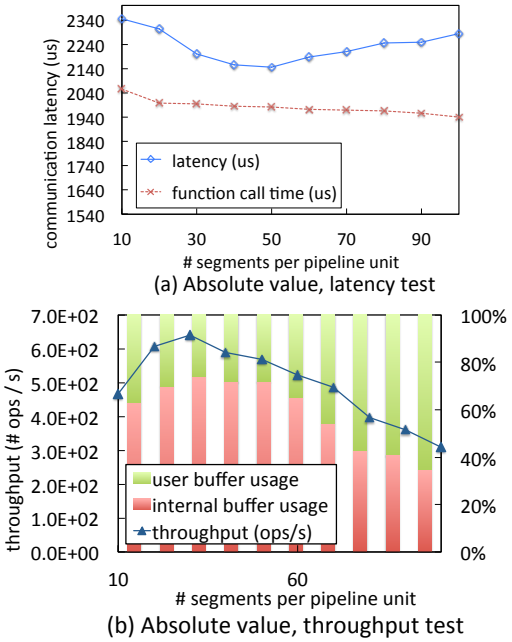

Figure 2: Communication latency and operation throughput with different numbers of segments per AM packet

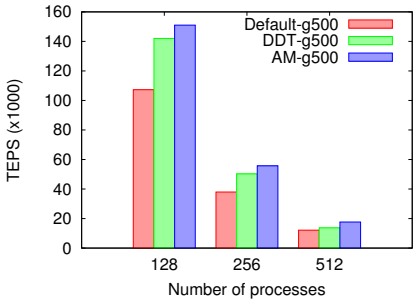

Figure 3: Graph500 comparative performance results

Figure 3 compares the performance of the AM implementation (AM-g500), derived datatype implementation (DDT-g500), and default implementation (Default-g500). Both AM-g500 and DDT-g500 coalesce messages. The AM-g500 performs better than DDT-500 because of the reduced amount of work when implemented with AMs.

III. Dynamic Task Parallelism

The second part of the Ph.D. work focuses on supporting dynamic task parallelism from MPI runtime: a form of parallelization that can invoke simultaneous execution across different cores working on different functionalities of an application. The concept of "task" represents an asynchronous operation that specifies certain "computation + data." This is similar to what AM is doing. By issuing AMs, one process can assign multiple tasks on other processes. Furthermore, each task is free to spawn other tasks by issuing AMs from the AM handler.

A. Making MPI Calls from the AM Handler

To support spawning tasks from one task, MPI-AM must be extended to support invoking MPI calls from the AM han-

dler. An important challenge is how to guarantee the thread safety between the main thread and asynchronous threads for AMs, while minimizing the overhead. This challenge arises because an MPI implementation may choose to spawn one or more dedicated threads for processing incoming AMs; hence, the runtime may need to always raise the thread safety to the highest level, thereby incurring significant overhead.

One can allow the user to disable asynchronous processing so that the runtime needs to raise the thread safety only to the lowest level, but that will sacrifice concurrency. A better solution is to allow the user to pass hints to the runtime saying how the MPI calls will be made, for example only in the main program or in both the main program and AM handlers, or whether the MPI calls will be invoked concurrently, so that runtime can raise the thread safety to the proper level without always pushing to the safest but most expensive level. Enabling MPI calls from the AM handlers can also help the upper-layer runtime, such as fully supporting AMs in Coarray Fortran on top of MPI [15].

B. Dependent Execution of Tasks

To specify dependent execution of tasks, we need to leverage the request-based operations in MPI. This approach allows the user to associate a request object to an MPI call, so that the execution of another MPI call can depend on the completion of the request-based routine by checking the corresponding request. Furthermore, one task can wait on several specific tasks. By doing so, data movement can be triggered after certain computation is finished, and computation on different processes can be ordered. This approach relieves the user from explicitly specifying such orderings by themselves in the program, and the user can handle much more flexible relations among computations and data movement patterns.

C. Data Locality and Load Balancing

Given such an MPI-AM framework with enabling MPI calls from an AM handler and dependent execution of tasks, one interesting question is, Can we manipulate the location of executing tasks in order to improve the load balance and concurrency? If so, what is the performance impact for different choices of locality?

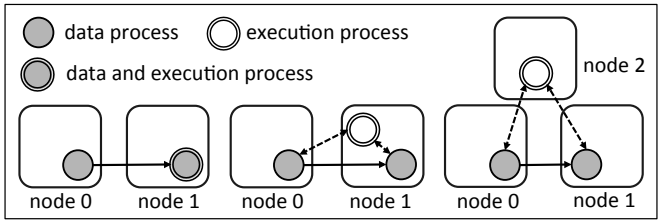

Figure 4: Different choices of MPI-AM data locality

As shown in Figure 4, for one task, some processes can be seen as *data processes*, which carry data related to that task, and some processes can be seen as *execution processes*, which execute the computation in that task. Data processes and execution processes need not be in the same set. Some tasks, such as those involving **MPI_COMM_RANK**, must be executed on data processes. Some tasks can be executed on other processes but within the same node, for example on a helper process that can directly access a data process's shared-memory region. Furthermore, some tasks can be executed on processes on a different node, by making the execution process fetch data locally. We would like to investigate the impact for different choices of data locality.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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.

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

[3] R. Harrison, G. Fann, T. Yanai, and G. Beylkin, "Multiresolution Quantum Chemistry in Multiwavelet Bases," *Springer Lecture Notes in Computer Science*, pp. 103–110, 2003.

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

[5] 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 ISCA*, New York, NY, USA, 1992.

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

[7] X. Zhao, P. Balaji, W. Gropp, and R. Thakur, "MPI-Interoperable Generalized Active Messages," in *Proceedings of ICPADS*, 2013.

[8] X. Zhao, P. Balaji, W. Gropp, and R. Thakur, "Optimization Strategies for MPI-Interoperable Active Messages," in *Proceedings of ScalCom*, 2013.

[9] 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 CCGRID*, 2013.

[10] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Proceedings of PACT*, 2010.

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

[12] 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.

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

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

[15] C. Yang, W. Bland, J. Mellor-Crummey, and P. Balaji, "Portable, MPI-Interoperable Coarray Fortran," in *Proceedings of PPoPP'14*. New York, NY, USA: ACM, 2014, pp. 81–92.