RUNTIME SUPPORT FOR IRREGULAR COMPUTATION IN MPI-BASED APPLICATIONS

BY

XIN ZHAO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor William D. Gropp, Chair
Professor Laxmikant V. Kalé
Professor Marc Snir
Dr. Pavan Balaji, Argonne National Laboratory

Abstract

In recent years there are increasing number of applications that have been using irregular computation models in various domains, such as computational chemistry, bioinformatics, nuclear reactor simulation and social network analysis. Due to the irregular and data-dependent communication patterns and sparse data structures involved in those applications, the traditional parallel programming model and runtime need to be carefully designed and implemented in order to accommodate the performance and scalability requirements of those irregular applications on large-scale systems.

The Message Passing Interface (MPI) is the industry standard communication library for high performance computing. However, whether MPI can serve as a suitable programming model / runtime for irregular applications or not is one of the most debated aspects in the community. The goal of this thesis is to investigate the suitability of MPI to irregular applications.

This thesis consists of two subtopics. The first subtopic focuses on improving MPI runtime to support the irregular applications from perspective of scalability and performance. The first three parts in this subtopic focus on MPI one-sided communication. In the first part, we present a thorough survey of current MPI one-sided implementations and illustrate scalability limitations in those implementations. In the second part, we propose a new design and implementation of MPI one-sided communication, called ScalaRMA, to effectively address those scalability limitations. The third part in this subtopic focuses on various issuing strategies in MPI one-sided communication. We propose an adaptive issuing strategy which can adaptively choose between delayed issuing strategy and eager issuing strategy in MPI

runtime to achieve high performance based on current communication volume in MPI-based application. The last part in this subtopic is to tackle the scalability limitations in the virtual connection (VC) objects in MPI implementation. We propose a scalable design to reduce the memory consumption of VC objects in MPI runtime.

The second subtopic of this thesis focuses on improving MPI programming model to better support the irregular applications. Traditional two-sided data movement model in MPI standard designed for scientific computation provides a paradigm for user to specify how to move the data between processes, however, it does not provide interface to flexibly manage the computation, which means user needs to explicitly manage where the computation should be performed. This model is not well suited for irregular applications which involve irregular and data-dependent communication pattern. In this work, we combine Active Messages (AM), an alternative programming paradigm which is more suitable for irregular computations, with traditional MPI data movement model, and propose a generalized MPI-interoperable Active Messages framework (MPI-AM). The framework allows MPI-based applications to incrementally use AMs only when necessary, avoiding rewriting the entire MPI-based application. Such framework integrates data movement and computation together in the programming model and MPI can coordinate the computation and communication in a much more flexible manner. In this subtopic, we propose several strategies including message streaming, buffer management and asynchronous processing, in order to efficiently handle AMs inside MPI. We also propose subtle correctness semantics of MPI-AM to define how AMs can work correctly with other MPI messages in the system, from perspectives of memory consistency, concurrency, ordering and atomicity.

To my parents, for their endless love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

As we move from the current multi-petaflop machines to Exascale computing, the primary constraint that impacts all aspects of hardware, software and design of application is the power usage of the machine. Data movement is the main consumer of the power and the community believes that this trend will not change. Consequently, applications and system software are reacting by migrating from traditional regular computational models involving regular data structures such as dense matrix to more sparse computational models involving irregular computational and communication patterns.

Irregular computation and communication models have already gained significant importance in recent years. Such models are widely used in applications from various domains such as bioinformatics (SWAP [1], Kiki [2]), computational chemistry (MADNESS [3] [4], NWChem [5]), and graph algorithms in social network analysis. Newer applications in other domains such as material science and nuclear reactor research are also looking to investigate more irregular computational models to improve their data movement efficiency ([6] [7]).

Irregular models differ from traditional computational models in many ways. For example, they are often organized around sparse structures such as sparse matrices or graphs rather than dense matrices. Similarly, their data movement is often irregular and data-driven rather than relying on a static predictable communication pattern. Furthermore, the growth rate of data movement cost with respect to system size or problem size is typically significantly

higher than that of the computation cost. Given the trend towards irregular computations, there is a large debate in the community with respect to whether traditional communication models that have been designed for more traditional communication patterns would be suitable to irregular computation.

Message Passing Interface (MPI) [8] is the industry standard communication library for high performance computing, with implementations available on virtually every parallel system in the world. Despite MPI's great success in high performance computing, whether it can serve as a runtime system for irregular applications or not is one of the most debated aspects in the community, especially as we move to exascale systems.

Some researchers believe that MPI can sufficiently well support higher-level programming models and runtimes that support irregular computations. For instance, there are several tasking models and irregular applications in the literature: ADLB [9], Scioto [10], NWChem [5], MADNESS [3]. Other researchers, however, doubt MPI's ability to succeed in such environments due to several reasons:

- **Scalability:** MPI has traditional been viewed as a communication library suitable (or "optimized") for regular or structured communication patterns such as stencil computations or distributed dense matrix computations. However, as we move to irregular communication patterns, such as those used within graph-centric applications (e.g., in the bioinformatics domain), researchers wonder if MPI is still a suitable runtime system. A commonly mentioned example in the literature is the failure of MPI in scaling to large problem sizes with the Graph 500 benchmark [11]. Specifically, most MPI implementations run out of internal resources when scaling the Graph 500 benchmark on large systems due to internal resource management issues within the MPI implementation.

- **MPI communication semantics:** The MPI standard provides mechanisms to move data using several communication patterns, including two-sided (MPI_SEND, MPI_RECV), collective (MPI_BARRIER, MPI_BCAST) and one-sided communication. Even

though there are different communication patterns being provided in MPI, there is still a concern that MPI solely focuses on the mechanisms for data movement and leaves the control of what data needs to be moved (e.g., whether it is better to fetch data and compute locally or send data and trigger computation remotely) to a higher-level runtime system or application. Some people's opinion is that this is the responsibility of high-level runtime systems and should not be handled by MPI library, whereas other people believe that MPI could absorb a more data-driven communication style (such as active messages) into its communication semantics.

In summary, despite the debate in the community, we do not believe that the suitability of MPI for irregular applications is a black-or-white question. Rather, we believe that there exists a whole spectrum of grayscale, where MPI would lie. The goal of this thesis is not to prove or disprove either point of view, but to perform a detailed study as to where in this spectrum MPI lies. The idea of this thesis is to influence future MPI implementations and standards to understand the implications of irregular applications and propose what if anything needs to change to efficiently support such applications.

1.1 Plan of Study

The thesis focuses on two pieces. The first part focuses on improving the MPI implementation to better support irregular applications through improved scalability and performance. The second part focuses on potential extensions to the MPI standard to be more data-driven allowing for simultaneous migration of computation and data in an "active-messages" like model.

1.1.1 Improvements to MPI Implementations for Irregular Applications

Current MPI implementations suffer from several shortcomings and challenges in the context of irregular computations. One challenge is scalability. Irregular computation involves large number of outgoing asynchronous operations and MPI runtime needs to maintain the states for all of them. This can potentially cause the runtime to consume most of internal resources and leave no resources for the application. A scalable and sustainable resource management strategy that can maintain internal resources meanwhile not sacrificing the performance too much is necessary. Furthermore, since one process can communicate with many other peers concurrently in the application, the runtime has to maintain the states for all possible communication peers which potentially can cause $O(P^2)$ memory consumption. With the rapid growth of the size of HPC systems in the world [12], steps must be taken to avoid linear growth of memory consumption with respect to system scale.

1.1.2 Improvements to MPI Standard for Data-driven Computations

The second part of the thesis focuses on extending the MPI programming model to better support the irregular applications. Traditional programming approaches that were designed for environments where computation is regular and its cost is typically significantly larger than the data movement cost are not well suited for irregular computation. The Active Messages (AM) model [13] is an alternative parallel programming paradigm that can potentially bridge this gap. It allows the sender to move a small piece of data to the receiver and to trigger computation upon arrival of the data, without the receiver explicitly receiving the data. Such a model can be more natural to use in some, though not necessarily all, scenarios in irregular computation. A generalized framework is needed which can combine both traditional MPI and AM capabilities in the MPI runtime, so that an application can be modified

incrementally to use AMs only when necessary, avoiding being rewritten entirely. Given such a framework, interesting questions rise including: How to make AMs work correctly with MPI infrastructure? How to optimize the performance of the framework in different application scenarios? And since AM is working in an one-sided manner, what is the performance impact for an asynchronous progress engine to process the incoming AMs? In this work we propose a generalized MPI-interoperable AM (MPI-AM) framework to address these issues.

1.2 Outline of the Thesis

The rest of this thesis is organized as follows: Chapter 2 presents the related background this thesis relies on; Chapter 3, Chapter 4 and Chapter 5 describe the problems in MPI runtime with respect to scalability and performance, and corresponding solutions; Chapter 6, Chapter 7 and Chapter 8 present the work on combining MPI with Active Messages model and discusses critical issues raised including message streaming and buffering, correctness semantics, performance optimization and asynchronous processing. Chapter 9 presents the conclusion of this thesis.

CHAPTER 2

Background

Because the work described in this thesis is largely relying on MPI communication paradigms, with an emphasis on the semantics of MPI one-sided communication, in this section we provide related background on MPI communication paradigms so that the reader can better understand the semantics MPI standard provides.

MPI standard provides various communication paradigms for upper layer applications and runtimes to use, including two-sided communication (e.g. MPI_SEND and MPI_RECV), collective communication (e.g. MPI_BARRIER, MPI_BCAST and MPI_REDUCE), and one-sided communication (e.g. MPI_PUT and MPI_GET).

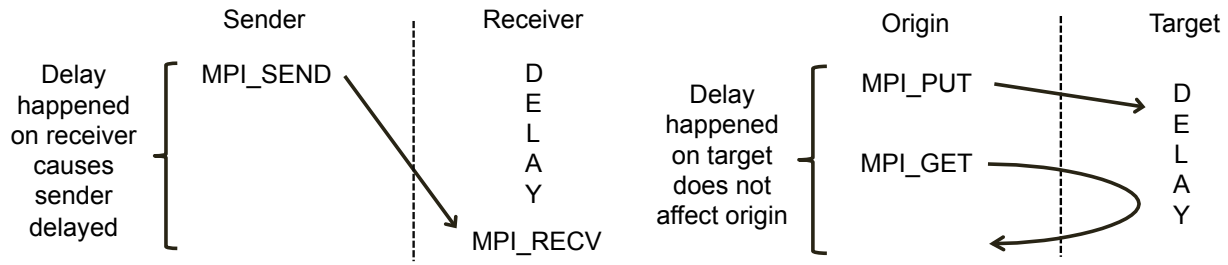
In MPI two-sided communication, both sender and receiver are required to explicitly participate in the communication: the sender invokes an MPI_SEND call, and the receiver invokes a corresponding MPI_RECV call. Only when MPI_SEND is completed on the sender, the data is safely sent out from the sender's buffer; similarly, only when the matching MPI_RECV is completed on the receiver, the data is safely received in the receiver's buffer. Collective communication coordinates date movement and synchronization among a group of processes in a communicator. Even though such communication can be implemented using two-sided calls, MPI provides convenient interfaces for user to directly perform such commonly used communication patterns (e.g. MPI_BARRIER, MPI_BCAST, MPI_REDUCE) and MPI implementations can implement those communication patterns using optimized collective algorithms.

Both two-sided and collective communication paradigm have been supported since MPI-1 standard. They combine data movement and synchronization together, and the programmer is clear about when communication is finished and when data in sender's and receiver's buffers can be safely accessed. Even though theoretically every MPI application can be implemented by using two-sided communication, in practice such an implementation brings significant overhead. The two-sided communication requires that the parameters in MPI_SEND and MPI_RECV to be matched with each other. Consequently, the programmer must plan for data layout and communication pattern on both the sender side and receiver side—a significant inconvenience in writing the application where data access patterns are not determined but dynamically change throughout the execution, resulting in complex code structure. On the other hand, the receiver must explicitly poll for potential incoming messages, and the runtime system must guarantee the ordering of two-sided messages, which incurs significant overhead to the application.

In order to overcome these issues, MPI-2 standard added a new communication paradigm, called one-sided communication, also known as remote memory access, or RMA. One-sided communication is a message passing paradigm that allows one process, the "origin" process (i.e., the process who initializes one-sided operations), to specify all communication parameters, for both the local side and the remote side; the "target" process (i.e., the process who is remotely accessed by origin process) does not need to explicitly make any call in order to processing incoming one-sided operations. Typical example of communication calls includes MPI_PUT and MPI_GET. One-sided communication was extended in MPI-3 with richer functionality and better accommodation with the latest hardware.

Compared with two-sided communication, one-sided communication decouples data movement and synchronization. The runtime system is able to move the data without requiring the remote process to synchronize, and the delay on the receiver process does not effect the process on the sender, as illustrated in Figure 2.1. This capability is achieved by having every process expose a part of its memory to the other processes from the beginning. The

sender process can directly read from or write to this memory, and processes involved in the communication define the synchronization points before one-sided communication begins and after it ends.

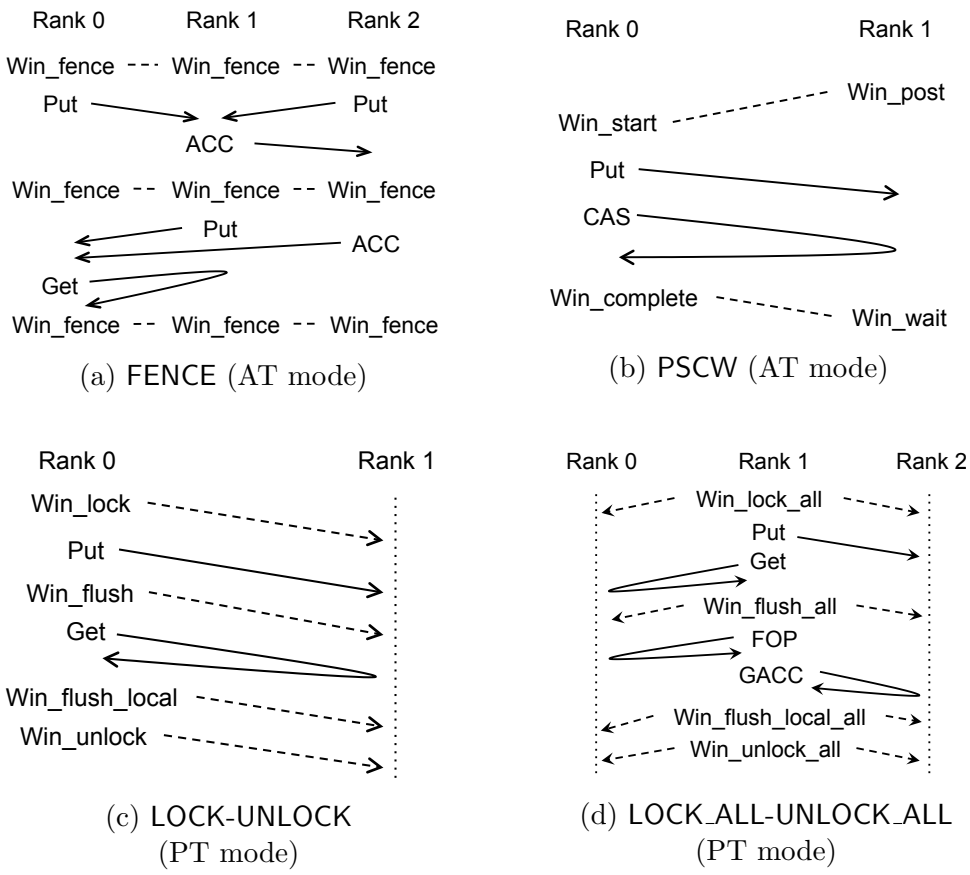


(a) Two-sided communication (b) One-sided communication

Figure 2.1: Comparing one-sided and two-sided communication paradigms

In order to allow the origin process to directly access the data on the target side, the target process needs to first expose a memory region, called a "window", beforehand, which is remotely accessible by all other processes. A window may be created in four ways: MPI_WIN_CREATE, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_CREATE_DYNAMIC. MPI_WIN_CREATE requires the user to pass an existing memory region to the function call and creates a window in that region; MPI_WIN_ALLOCATE allocates a memory region by itself and creates a window in it; MPI_WIN_ALLOCATE_SHARED is similar to MPI_WIN_ALLOCATE except that it is used by processes within the same node and creates a window in a shared-memory region among those processes; MPI_WIN_CREATE_DYNAMIC creates an empty window and allows the user to dynamically attach memory buffers to the window afterwards.

After a remote accessible window has been defined, the origin process needs to initialize a one-sided communication "epoch" in which the one-sided communication occurs. The epoch defines when one-sided operations can be issued and when modifications by one-sided operations are completed on the origin and target processes. Two synchronization modes—"Active Target (AT)" mode and "Passive Target (PT)" mode—are defined in the MPI standard and must be called at the beginning and end of a one-sided communication epoch. Figure 2.2



(a) FENCE (AT mode)

(b) PSCW (AT mode)

(c) LOCK-UNLOCK (PT mode)

(d) LOCK_ALL-UNLOCK_ALL (PT mode)

Figure 2.2: Synchronization modes in MPI RMA interface

illustrates how those synchronization modes work. The Active Target mode requires that both the origin and the target processes invoke the synchronization calls. MPI offers two types of Active Target synchronization: FENCE and POST-START-COMPLETE-WAIT (PSCW). The Passive Target mode only requires the origin process to make the synchronization call, which also contains two types: LOCK-UNLOCK, and LOCK_ALL-UNLOCK_ALL. Note that there are two types of locks defined in the PT mode: (a) MPI_LOCK_SHARED: multiple process can acquire a lock on the same target concurrently; (b) MPI_LOCK_EXCLUSIVE: only one process can acquire a lock on the target process.

MPI defines six types of data movement operations: MPI_PUT, MPI_GET, MPI_ACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_COMPARE_AND_SWAP. MPI_PUT transfers the data from the origin process to the target window. MPI_GET transfers the data from the target window to the origin pro-

cess. MPI_ACCUMULATE combines the data from the origin process with the data on the target window by triggering certain predefined computations on the target process. MPI_GET_ACCUMULATE is similar to MPI_ACCUMULATE except that it returns the original value of the target data. MPI_FETCH_AND_OP is a special case of MPI_GET_ACCUMULATE in which it accumulates only one basic element from the origin buffer to the target buffer. In this thesis we call MPI_ACCUMULATE, MPI_GET_ACCUMULATE, and MPI_FETCH_AND_OP as "MPI_ACCUMULATE-like operations", since all of them involves an computation to accumulate origin data and data on the target side. MPI_COMPARE_AND_SWAP also moves one basic element from the origin buffer to the target buffer. However, it moves two elements: the origin element and the compare element. It first compares the compare element with the target element; if they are identical, it replaces the target element with origin element and returns the original target value. MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP and MPI_COMPARE_AND_SWAP perform a "read-modify-write" style of one-sided operation between the origin and target processes.

CHAPTER 3

Tackling Scalability Challenges in MPI One-Sided Infrastructure

Irregular applications often involve massive outstanding asynchronous messages, in which each process can potentially communicate with large amount of peers. Such characteristics make the scalability and performance of handling massive asynchronous communication in MPI runtime to be crucial for the performance of the irregular applications. In this chapter, we describe the challenges existing in the current MPI runtime, with respect to scalability and performance in two aspects: MPI one-sided communication and initialization of virtual connection objects. We propose corresponding solutions to make the MPI runtime be capable of addressing those challenges in order to efficiently support irregular applications on large scale systems.

MPI one-sided communication, also known as Remote Memory Access (RMA), is becoming increasingly popular in many applications particular those that rely on irregular communication patterns. One-sided communication provides a different execution model, compared to traditional two-sided or group communication, which is better suited for some classes of applications. However, current implementations of MPI one-sided communication are notorious for their inability to scale to large systems or problem sizes. In this section, we first present a thorough study of various MPI one-sided communication implementations, including their strengths and shortcomings. Next, we present a new MPI one-sided communication

system, called ScalaRMA, that addresses the scalability problems of these implementations and provides a highly scalable MPI one-sided communication runtime. The driving goal of ScalaRMA is to achieve perfect scalability with respect to constant maximum resource usage, irrespective of problem or system size. We present our investigation and show that doing so with little to no performance degradation is possible in most, but not all, cases. In cases where this is possible, we present a detailed analysis of the workings of ScalaRMA, including a thorough performance evaluation and profiling information internal to the MPI implementation. In cases where this is not possible, we present aspects within the MPI-3 standard and / or the network hardware implementation that make this challenging.

3.1 Overview

The MPI Forum introduced one-sided or remote memory access (RMA) communication, as an alternative to the traditional two-sided and group communication operations, in MPI-2. In MPI-3 [8], the RMA capabilities of MPI went through a significant revamp leading to newer, cleaner, and more performance-capable RMA semantics. In the RMA model, one process (i.e., the origin) can directly and implicitly access the memory on another process (i.e., the target) without requiring any explicit communication calls from the target. Such a model provides a different execution paradigm than traditional two-sided or group communication, making it an attractive alternative for some applications.

In the recent past, several applications, particularly those with irregular communication patterns, have started investigating MPI RMA as an alternative to their existing usage of MPI. Such a move is motivated primarily by the fact that MPI RMA does not require processes to be "cooperative"; that is, the origin does not need to explicitly match a call on the target. This approach improves the ease of writing applications especially when communication is irregular and data-driven, because the programmer no longer needs to carefully plan the communication pattern. For example, in NWChem [5], a large quantum chemistry

application, massive asynchronous messages need to be communicated and a process typically does not know whom to receive the message from. Thus, it uses RMA to implement a model where each process can dynamically fetch data to local memory, compute, and write the computed output to a target remote memory region.

Despite its growing prominence, current implementations of MPI RMA are notorious for their inability to scale to large systems or problem sizes. A commonly mentioned example in the literature is the failure of MPI to scale to large problem or system sizes with the Graph 500 benchmark [11]. Specifically, most MPI implementations run out of internal resources when scaling Graph 500 to large problem or system sizes. Irregular communication, such as that evidenced in Graph 500, involves a large number of outgoing asynchronous operations, and the MPI runtime needs to maintain the state of each one of them. This requirement can potentially cause the runtime system to consume large amounts of internal resources and leave no resources for the application. Clearly needed are scalable and sustainable resource management strategies that can manage internal resources. Similarly, since each process can communicate with many other peers in the application, the runtime has to maintain state for all possible communication peers, a situation that potentially can cause $O(P)$ memory consumption.

While significant research has been done on improving MPI RMA, such work is fragmented over a number of different MPI implementations. Such fragmentation makes it hard to create a baseline of comparison for additional improvements that we will propose in this section. Therefore, before we discuss any new research contributions of this section, we first present a thorough study of various MPI RMA implementations. In particular, we survey the strengths and shortcomings of these implementations and consolidate the best of these capabilities into a common "state-of-the-art" baseline MPI implementation. Next, we present a new highly scalable MPI RMA implementation, called ScalaRMA, that addresses the scalability problems of these implementations. Our experimental evaluation compares ScalaRMA with the consolidated baseline MPI implementation.

The primary objective of ScalaRMA is to achieve perfect scalability with respect to constant maximum resource usage, irrespective of problem or system size. Through the course of our investigation, however, we identified that doing so with little to no performance degradation is possible in most, but not all, cases. In cases where such perfect scalability is not possible, we present aspects within the MPI standard and / or the network hardware implementation that make achieving this goal challenging. We also propose potential future enhancements to network hardware to alleviate some of these issues.

3.2 Survey of RDMA Capabilities on Modern Networks

Remote direct memory access (RDMA) for network hardware is a common and significant feature occurred on most modern network architectures today. However, each network provides these capabilities in a slightly different manner, making it important for upper layer runtime to understand those differences and design suitable algorithms for each network. In this section, we present a survey of RDMA capabilities on six different networks: Mellanox InfiniBand [14] [15], Portals-4 on Bull BXI [16] [17], the Tofu network architecture [18], the Cray Aries [19], the IBM Blue Gene/Q network (BG/Q) [20] and RDMA over Converged Ethernet (RoCE) [21]. For Mellanox InfiniBand, our survey is based on the ConnectX-4 hardware. Portals-4 is technically not a hardware specification but an API specification, and currently no hardware implements it. However, several network hardware implementations of Portals-4 have been announced, such as the BXI interconnect from Bull. Our survey is based on the available open literature on BXI. The Tofu network architecture is based on Fujitsu's K computer and the follow-on Fujitsu FX100 computer [22]: these are two separate generations of the Tofu network, for the purposes of this survey we consider them as the same. For Cray Aries, our survey is based on the Cray XC30 system used on NERSC Edison. BG/Q is the third generation in the IBM Blue Gene line of massively parallel supercomput-

ers and our survey on IBM network is based on it. RoCE is a network protocol that allows RDMA over an Ethernet network; our survey is based on the RoCE version 2 protocol. Of the various capabilities hardware capabilities provided by these networks, three out of them are of particular interest to us in this work of achieving scalability: window address calculation, memory protection keys, and remote notification. In the following sections, we will introduce each of them.

3.2.1 Window Address Calculation

The calculation of window address in communication operations of MPI RMA are often based on a model of "scaled offset". This means that the user on the origin side provides an offset within the window buffer on the target side and a "scaling factor" (or called displacement unit) that determines where the data needs to be written to or read from. Note that the offset is in scaling factors. Such a model, however, does not directly map to what current network hardwares provide. Some networks, such as Mellanox InfiniBand, accept only an absolute address to which data can be written to or read from: we classify them as "HW-addr"-based networks. In such cases, the MPI implementation needs to maintain the associated start or base address and scaling factor for each target process and translate the user-provided information to what the network hardware expects (absolute address). In order to do this translation, the MPI implementation needs to maintain $O(P)$ base addresses and scaling unit sizes.

Other networks, however, provide a richer model where the MPI implementation can directly provide the offset (in bytes) to the network hardware during an RDMA operation: we classify them as "HW-offset"-based networks. For such networks, MPI implementations no longer have to maintain an $O(P)$ data structure for the base addresses but do need to maintain an $O(P)$ data structure for the scaling factors. Fortunately, in practice, most applications provide the same scaling factor on all processes, thus requiring only $O(1)$ memory. Portals-4 is an example of such networks.

We can envision a network architecture that could allow the MPI implementation to directly provide the offset in units of the scaling factor, even though such hardware does not exist today. We classify such future network architecture as "HW-scalad-offset"-based networks. Given such network hardware, the MPI implementation no longer needs to maintain any $O(P)$ data structure for window address calculation at all. Such architecture is listed here because of completeness, but we do not propose any new techniques based on it in this work.

3.2.2 Memory Protection Keys

Occurrence of RDMA in network architecture allows multiple processes to directly access the memory region on the target side. While it is a very convenient feature for the upper layer runtime and application to use, it also brings a security concern. Considering secure HPC environments where multiple users may share the same supercomputer or the non-HPC environments where multiple users may share the same node, in those cases, protection between users is required. In order to guarantee the security among different users, some networks provide a scheme where the target side generates a protection key for its own memory, and only processes that have access to the corresponding protection key can access the memory region of that target.

Mellanox Infiniband is an typical example of such kind of networks, where a memory protection key, called an "remote key (rkey)", is required for the origin side in order to access a location on memory region with RDMA on target side. Furthermore, this key is asymmetric. This means that each process may generate a different key, therefore the origin side has to store a key for each potential target, which results in potential $O(P)$ memory usage. Other networks, like Tofu, requires no keys to access the memory region on target. Even though it is scalable, such network requires separate security protection scheme against unauthorized accesses.

3.2.3 Remote Notification

As we introduced before, RDMA operations are performed by the network hardware on target memory region without any involvement of the processor on target side. While this model is true for all networks, their notification mechanism is different with each other. Some networks, such as Mellanox InfiniBand, Cray Aries and IBM BG/Q, provide origin-side notification for both reading operations (GET) and writing operations (PUT), but target-side notification only for writing operations (PUT). Other networks, such as Portals-4 and Tofu, provide both origin-side notification and target-side notification for all kinds of RDMA operations.

This subtle difference in remote notification has a significant impact on how one MPI implementation would design synchronization algorithms for MPI RMA runtime. Specifically, if the network hardware provides target-side notification, the MPI implementation can design an algorithm where the target side can count the number of operations that are issued on it, therefore the origin side can avoid wait for remote completion for issued operations. More details of such designs are described in Section 3.4.

3.2.4 Summary of Network RDMA Capabilities

Table 3.1 summarizes the different hardware capabilities provided by each of the networks surveyed, from perspectives of window address calculation, memory protection keys and remote notification.

Table 3.1: Different capabilities provided by modern network hardwares

Network	**Window Address Calculation**	**Protection Keys**	**Remote Notification**
InfiniBand	HW-addr	Yes	PUT
Portals-4	HW-offset	No	PUT / GET
Tofu	HW-addr	No	PUT / GET
IBM BG/Q	HW-offset	No	PUT
Cray Aries	HW-offset	No	PUT
RoCE	HW-addr	Yes	PUT

3.3 State-of-the-Art of MPI Implementations and Scalability Challenges

Most MPI implementations have several areas, in aspects of window creation, synchronization and communication, that use nonscalable data structures, and there is no upper bound for the internal resources consumed by those data structures. This makes MPI runtime tend to use up all internal resources when problem size or system scale is large, not say leaving any resources for the application. In those cases, the application even cannot finish running. MPI runtime needs to be carefully investigated and re-designed in order to guarantee the sustainable and scalable execution of the application, meanwhile avoids sacrificing the performance too much. Here we describe the scalability challenges we observed when investigating existing MPI implementations.

Existing MPI implementations have several scalability limitations in their RMA infrastructure. Some of these limitations are common in all implementations while some of them are not the common issue but exist in one or two MPI implementations. In this section, we first introduce different implementation choices for MPI RMA operations, and then we investigate three existing open-source MPI implementations: MPICH (3.1.4) [23], Open MPI (1.10.2) [24], and MVAPICH (2.2b) [25]. The goal of this section is to identify the best choices of the different capabilities needed within MPI RMA and to consolidate them into a common "state-of-the-art" baseline of MPI implementation. The baseline will be referred to as "MPI-RMA-base" in the rest of this thesis.

3.3.1 Implementation Choices for MPI RMA Operations

Two implementation choices exist for MPI RMA operations in existing MPI RMA implementations: hardware-based (HW-based) operation, which is implemented by directly using hardware RDMA functionality, and active-message-based (AM-based) operation [13], in

which a software handler is invoked on the target process to execute the incoming operation.

- **HW-based operation**: it typically can achieve higher performance than do AM-based implementation but are restrictive. Most networks implement simple RDMA communication (such as writing or reading contiguous data) in hardware. But more complex communication such as that involving accumulating noncontiguous data segments is often not implemented in hardware.

- **AM-based operation**: it works as follows. An origin process issues a one-sided operation using a nonblocking `SEND` operation; a set of message handlers is predefined on the target process; and there is a receiving progress on the target process to process incoming messages (progress is triggered whenever the target process makes an MPI call). When an one-sided operation arrives, a corresponding message handler will be triggered in the receiving progress to process that operation.

Since HW-based operations is restrictive, most MPI implementations use a combination of HW-based and AM-based implementations where an operation uses hardware RDMA when available and otherwise falls back to AMs. We note that RMA between processes on the same node is often implemented by using direct memory accesses to a shared-memory region. We treat such operations as HW-based operations.

3.3.2 Window Creation

In MPI RMA interface, each RMA window is created with its own base address, window size, scaling factor size, and window handle. In MPI one-sided communication, the origin process needs to quickly access such information about remote processes in order to issue data movement operations. Most MPI implementations locally stores those information for all ranks on the window which requires $O(P)$ memory consumption per window on each process.

Window creation is the "setup phase" where memory region on target processes are made available for remote accesses. Like introduced in Chapter 2, MPI provides four ways of creating a window: MPI_WIN_CREATE, MPI_WIN_ALLOCATE, MPI_WIN_CREATE_DYNAMIC and MPI_WIN_ALLOCATE_SHARED. In each of these window creation models except for MPI_WIN_CREATE_DYNAMIC, the user needs to specify a scaling factor (i.e., displacement unit) and window base address (or window creation call generates a base address by itself), which is used to calculate the location of the data an RMA operation would access. Window creation calls are collective. During this phase, the MPI implementation needs to exchange necessary metadata information between different processes, allowing them to access memory region among each other. This metadata includes user-specified information such as the start address of the target buffer on the window and size of scaling factor, as well as network-hardware-specific metadata such as memory protection keys, as introduced in Section 3.2.2. The amount of memory required for storing such metadata information is the focus of this section.

As mentioned in Section 3.3.1, RMA operations are typically implemented as a combination of HW-based and AM-based operations. These different kinds of operations have unique characteristics as to what metadata they require. Specifically, AM-based operations require neither the user-specified nor the hardware-specific metadata for the target at the origin process. The origin process can simply send the origin-side data in the active message, while the active message handler that executes on the target process can look up the target-specific information on the target side. Open MPI, for example, implements AM-based operations in this way, thus using $O(1)$ memory for the associated metadata. MPICH and MVAPICH, on the other hand, store the window base address for each target process even for AM-based communication, thus using $O(P)$ memory for the associated metadata. We consider the Open MPI implementation as the state-of-the-art in this context, and adopt that model for the consolidated MPI-RMA-base implementation.

HW-based operations are more efficient than AM-based operations but can require more

metadata to be kept track of. Specifically, if the network hardware requires "HW-addr"-based communication (like Mellanox InfiniBand), the MPI implementation needs to keep track of the window base addresses of each of the target processes, thus requiring $O(P)$ storage for the corresponding metadata. Similarly, if the network hardware requires an asymmetric memory protection key for each target process (like Mellanox InfiniBand), that needs to be kept track of at the origin as well, again requiring $O(P)$ storage for the metadata. All three MPI implementations that we surveyed (Open MPI, MPICH, and MVAPICH) use $O(P)$ memory in this case, which we adopt for the consolidated MPI-RMA-base implementation.

Such metadata, however, is not required on network hardware that requires "HW-offset"-based communication (e.g., Portals-4). For such networks, the Open MPI implementation uses an $O(1)$ storage for the metadata. However, the MPICH implementation stores the window base address for each target process, thus effectively ignoring the "HW-offset" capability of the network and using an $O(P)$ storage for the metadata. We consider the Open MPI implementation as the state of the art in this context, and we adopt that model for the consolidated MPI-RMA-base implementation.

With respect to the user-provided metadata, if the user specifies different size of scaling factor for each process, such information also needs to be kept track of at the origin process as well, thus requiring $O(P)$ storage for the metadata. All three MPI implementations that we surveyed (Open MPI, MPICH, and MVAPICH) use $O(P)$ memory in this case, which we adopt for the consolidated MPI-RMA-base implementation.

These memory requirements are summarized in Table 3.2. AM-based operations should only require $O(1)$ memory since they do not need the origin to keep track of the user-specified or hardware-specific metadata. On the other hand, for HW-based operations, the amount of metadata required depends heavily on the capabilities of the network. Networks that require asymmetric memory protection keys require $O(P)$ memory. Networks that require "HW-addr" based communication only require $O(1)$ memory when the base address of the target window can be looked up efficiently (e.g., if the allocation is symmetric where all

Table 3.2: Baseline memory usage for window metadata

<table>
<tr><th rowspan="3">Window creation calls</th><th rowspan="3">AM-based operation</th><th colspan="4">HW-based operation</th></tr>
<tr><th rowspan="2">Asymmetric keys</th><th colspan="3">Symmetric or no keys</th></tr>
<tr><th>HW-offset</th><th>HW-addr</th><th>HW-scaled-offset</th></tr>
<tr><td>MPI_WIN_CREATE_DYNAMIC</td><td rowspan="6">$O(1)$</td><td rowspan="6">$O(P)$</td><td colspan="2">$O(1)$</td><td rowspan="6">$O(1)$</td></tr>
<tr><td>MPI_WIN_ALLOCATE_SHARED</td><td colspan="2">$O(P)^*$</td></tr>
<tr><td>MPI_WIN_CREATE + same scaling unit size</td><td>$O(1)$</td><td>$O(P)$</td></tr>
<tr><td>MPI_WIN_CREATE + different scaling unit size</td><td colspan="2">$O(P)$</td></tr>
<tr><td>MPI_WIN_ALLOCATE + same scaling unit size</td><td colspan="2">$O(1)$</td></tr>
<tr><td>MPI_WIN_ALLOCATE + different scaling unit size</td><td colspan="2">$O(P)$</td></tr>
</table>

* P is smaller than or equal to the number of processes within one node. We consider it as constant memory usage.

targets have the same base address); otherwise, their memory usage would go up to $O(P)$.

As discussed in Section 3.2, the MPI implementation and network interface need to translate the offset in scaling units in one-sided operations to an absolute address on the target window. This step requires either the MPI runtime or the network interface to use $O(P)$ memory in order to maintain the necessary window metadata: namely, the base addresses and sizes of scaling units. In some situations, however, such memory consumption can be completely avoided in the MPI runtime. First, for hardware that is HW-scaled-offset (as discussed in Section 3.2), the runtime system does not need to maintain any window metadata because no translation is required. Second, if all operations are AM-based operations, the runtime system does not need to maintain any metadata because the origin process can send the offset in scaling units to the target and the target can calculate the absolute address by itself. Third, both MPI_WIN_CREATE_DYNAMIC and MPI_WIN_ALLOCATE_SHARED use constant memory for window metadata.

When these conditions are not satisfied, the runtime system needs to maintain window metadata for all other processes. Nevertheless, in some specific situations, the runtime sys-

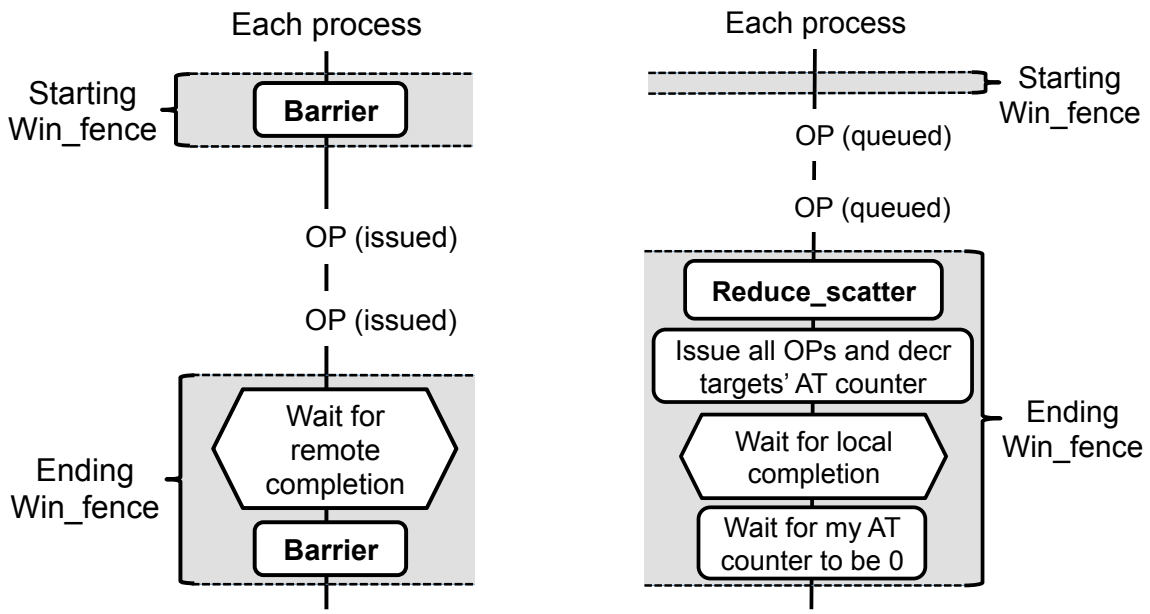
tem still can save $O(P)$ memory usage. Specifically, it does not need to always maintain base addresses. In the case of MPI_WIN_CREATE, either the runtime system or the hardware network needs to maintain base addresses in order to calculate absolute addresses. For the HW-offset network, which can maintain them by itself and accept offset in bytes as input argument, the runtime system does not need to maintain base addresses at all. When a window is created by MPI_WIN_ALLOCATE, memory usage of base addresses can be completely avoided by using a "symmetric allocation" strategy, which can generate the same base addresses on all processes.

3.3.3 RMA Synchronization

In this section, we describe the synchronization algorithms used in existing MPI implementations and scalability challenges in different algorithms.

Active Target Algorithms. We first discuss algorithms used in Active Target mode. The MPI-3 standard states that the return of the epoch-ending routine MPI_WIN_FENCE on the origin process guarantees that all operations issued are locally completed, whereas its return on the target process guarantees that all operations targeting that process are completed. Two algorithms are used in MPICH, Open MPI, and MVAPICH: the BARRIER-based algorithm and the REDUCE_SCATTER-based (RS-based) algorithm.

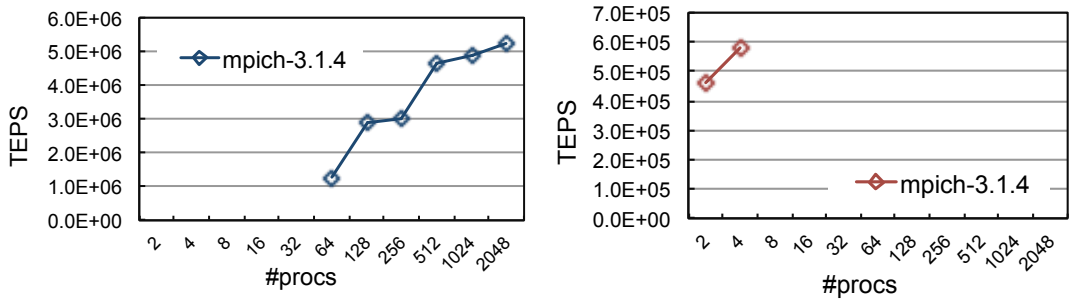
In the BARRIER-based algorithm (Figure 3.1a), starting MPI_WIN_FENCE performs a BARRIER among all processes on the window; all posted operations are issued immediately ("eager issuing"); the MPI_WIN_FENCE waits for remote completion for all issued operations and performs another BARRIER at the end. This algorithm is scalable. However, it imposes a stricter synchronization than what the MPI standard requires: it forces a starting call to be blocking, which is not required by the MPI-3 standard, and origin processes have to wait for remote completion, a requirement that is stricter than the local completion required in the MPI-3 standard. Existing MPI implementations use this algorithm for HW-based operations.



(a) BARRIER-based algorithm (b) RS-based algorithm

Figure 3.1: FENCE algorithms used in existing MPI implementations

In the RS-based algorithm (Figure 3.1b), starting MPI_WIN_FENCE does nothing and returns immediately; all posted operations are queued in the runtime system ("lazy issuing"); MPI_WIN_FENCE first performs a REDUCE_SCATTER to get the number of processes that will issue operations to it and initializes a local Active Target (AT) counter with that value. Next it issues all pending operations, in which the last operation to each target decrements the target's AT counter; then it waits for local completion of all issued operations and the local AT counter to be 0, which means all operations targeting it have been completed. This algorithm implements exact semantics as required by the MPI-3 standard and does not have extra synchronization overhead. However, it has two disadvantages. First, a REDUCE_SCATTER call requires a $O(P)$ data structure to be used. Second, since there is no synchronization at the beginning and all operations are queued until the end, the runtime system has to maintain an unlimited amount of operation metadata (issuing parameters). Existing MPI implementations use this algorithm for AM-based operations. If the implementation can utilize the remote side notification feature from hardware, this algorithm also can be used with HW-based operations.



(a) Strong scaling
(problem size: 2^{22})

(b) Weak scaling
(problem size: 2^{19} to 2^{29})

Figure 3.2: Graph 500 benchmark results
with RS-based algorithm

Figure 3.2 shows examples of running the Graph 500 benchmark with the RS-based algorithm. The benchmark is broken when the runtime system tries to maintain too much operation metadata.

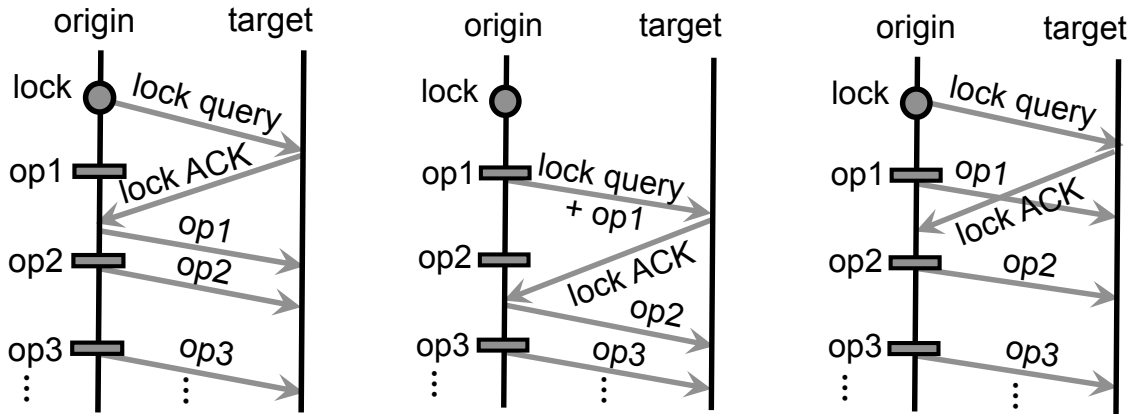
Passive Target Synchronization. In this section, we discuss the algorithms used in Passive Target mode, including maintaining lock state correctly in LOCK-UNLOCK and LOCK_ALL-UNLOCK_ALL and managing concurrent passive lock.

In Passive Target (PT) mode, the MPI runtime system needs to maintain a "target object" for each target to store two kinds of state: a flag indicating whether MPI_MODE_NOCHECK is set and an error-checking flag to prevent issuing two concurrent locks to the same target. On the other hand, if MPI_WIN_LOCK or MPI_WIN_LOCK_ALL is implemented as a nonblocking call, a target object is also needed to store state indicating whether a lock is granted. Such target objects require $O(P)$ memory in the worst case.

Several processes may try to acquire a lock on the same target simultaneously, which has already been granted exclusively to someone else. In such a case, those processes have to wait for the lock to be released and then compete for that lock. Open MPI uses a "network polling" strategy for HW-based implementations. In MPI_WIN_LOCK the origin process acquires a lock by repeatedly checking a memory location on the target process by issuing RDMA

operations (COMPARE_AND_SWAP) until the lock is granted; in MPI_WIN_UNLOCK the origin releases the lock by writing to that memory location using an RDMA PUT. This strategy generates considerable network traffic in the blocking call MPI_WIN_LOCK, and it cannot guarantee fairness among all competitors. On the other hand, MPICH and MVAPICH use a "queued locks" strategy. Each process issues a lock query message to a target process in MPI_WIN_LOCK; on that target process, a lock will be granted to the query that arrives first, whereas all other unsatisfied lock queries are queued up; when the current lock is released, the target process grants the lock to the next query in the queue. This strategy involves no extra traffic because each competitor issues only one message to the target, and it guarantees fairness among all competitors. However, there is no upper bound on how many lock queries will be queued on the target; in the worst case, it can be $O(P)$.

Two optimization are available with the queued locks strategy, as shown in Figure 3.3. One is "piggybacking locks (PB)": the origin merges the lock query with the first operation and issues it. Another is "speculative issuing (SI)": the origin speculatively issues the first operation without receiving acknowledgment of a granted lock; if the lock is not granted but is queued on the target, that operation is queued on the target also until the corresponding lock is granted. Queuing of speculative operations also leads to unlimited resource usage.



(a) No optimization (b) PB optimization (c) SI optimization

Figure 3.3: Optimization in queued locks strategy

3.3.4 Data Movement Operations

In this section, we first defines five states of RMA operation, then we describe the strategies used to issue HW-based operations and AM-based operations in existing MPI implementations and scalability challenges.

Operation States. After an RMA operation is posted on the origin, it can be in five states. The initial state is "not issued", in which synchronization has not been finished, operation cannot be issued, and the runtime system must maintain issuing parameters for that operation, including buffer address, count, datatype, and computation type (MPI_ACCUMULATE-like operations). When synchronization is finished, the runtime system can start to issue that operation. For a large MPI_ACCUMULATE-like operation, MPI implementations may stream it into multiple smaller units and issue them out one by one; in such cases, the operation state is switched to "partially issued". For units that have not been issued out, the runtime system needs to maintain issuing parameters; for units that have been issued out, it needs to maintain the metadata required for detecting local and remote completion. When the entire operation is issued out, the operation state is switched to "fully issued". In such cases, the runtime system no longer needs to maintain any issuing parameters, but it needs to maintain the metadata for local and remote completion. When local completion is detected—for example, data is completely issued out from the origin buffer in MPI_PUT—the operation state is switched to "locally completed". When remote completion is detected—for example, the origin process receives a notification from the target that all data in MPI_PUT has arrived on the target—the operation state is switched to "remotely completed", and the runtime system no longer needs to maintain any metadata for that operation. The state transition is shown in Figure 3.4.

As shown in Table 3.3, it is clear that runtime always needs to maintain metadata for every operation except for the final state. Whenever the operation state is changed, some

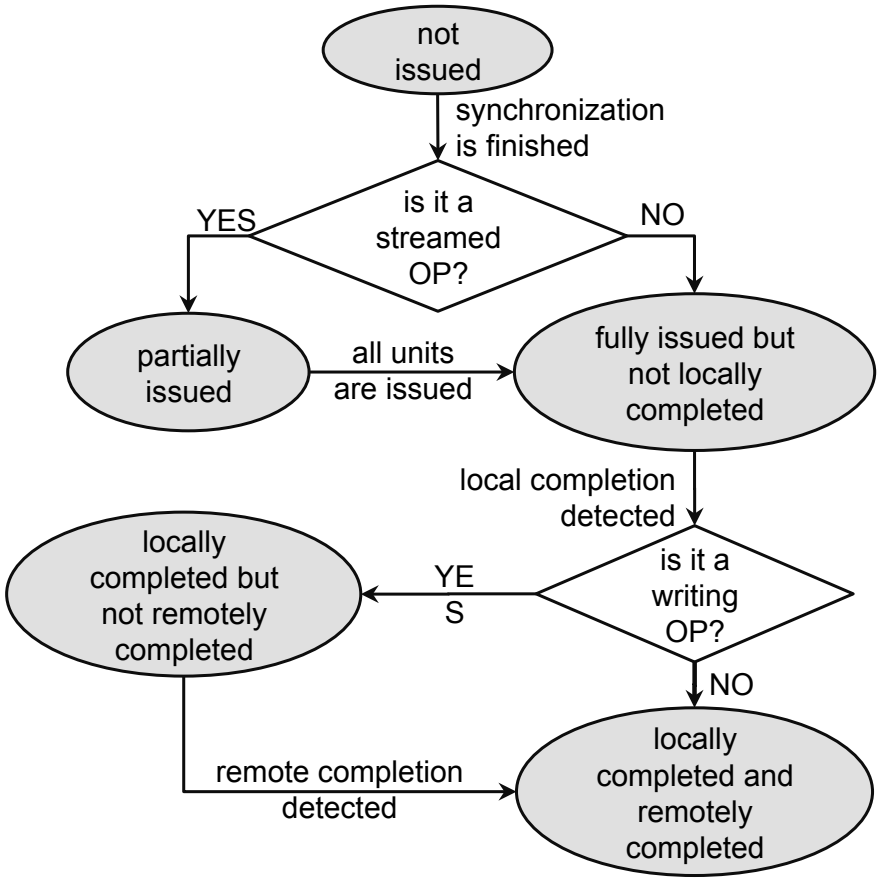


Figure 3.4: Operation states transition

metadata is no longer needed and can be dropped, but some still needs to be maintained by the runtime. Because there is no upper bound on how many operations will be posted in an application, the memory consumption of metadata can be very large and is likely to use up all internal resources of MPI runtime. A proper resource management strategy is required to keep the memory usage for operation metadata at a constant level.

Operation Issuing Strategies. In the irregular computation, one process usually issues large number of asynchronous operations. If those operations are not issued out by the runtime, or are issued but not completed immediately, the MPI runtime needs to maintain the states for them.

There are two strategies of issuing asynchronous operations in MPI runtime: one is de-

Table 3.3: Metadata needed in each operation state

Operation states	**Metadata**		
	Issuing parameters	**Metadata for local completion**	**Metadata for remote completion**
Not issued	✓	✗	✗
Partially issued	✓	✓	✓
Fully issued	✗	✓	✓
Locally completed	✗	✗	✓
Remotely completed	✗	✗	✗

laying issuing till the ending synchronization, in which case the runtime can save synchronization overhead by piggybacking synchronization message with RMA operations, and does optimization of coalescing operations. Another strategy, which is more naive with the RDMA operations, is to issue out the operation as soon as possible. Most current MPI implementations choose the first way to issue the RMA operations, which potentially causes large number of operations to be maintained in the runtime simultaneously.

Another issues is that, currently most MPI implementations implement RMA operations using software method, which essentially use SEND / RECV underlying to implement functionality of RMA. This enforces the MPI runtime to take charge of all outgoing operations and maintain the states of them until they are completed. Similar with the problem of issuing strategy, since there are no upper bound for how many operations are going to be issued in the application, the internal resources are very easy to be used up. When MPI is running on top of the hardware which provides the one-sided capability, MPI runtime should offload the handling of RMA operations to the hardware as much as possible and let the hardware to detect the completion state, releasing itself from the responsibility of maintaining all operations internally.

In Section 3.3.3, we mentioned two strategies for issuing operations in FENCE algorithms:

eager issuing and delayed issuing. These two strategies are used in other synchronizations [26] also. The eager issuing strategy issues every operation as soon as possible and does not need the runtime to maintain issuing parameters for each operation; however, it requires both starting and ending synchronization calls to be blocking. Lazy issuing stores issuing parameters for each operation in an "operation object" and queues up all operation objects in the runtime system until an ending call. By doing so, synchronization in the starting call can be avoided; but it can cause serious problem in memory usage because the number of queued operation objects can potentially use up all internal resources when the problem size is large. Existing MPI implementations use eager issuing for HW-based operations and delayed issuing for AM-based operations.

Metadata for Local and Remote Completion. As discussed above, after an RMA operation is completely issued out, the runtime system or hardware needs to maintain metadata in order to detect the local and remote completion for that operation. For HW-based operations, such metadata is maintained by hardware, and the runtime system does not need to care about it. For AM-based operations, the MPI runtime needs to use a "request object" to keep track of issued but incomplete operations. MPICH, Open MPI, and MVAPICH have no resource management strategy in the runtime system to manage request objects, which can easily use up internal resources; for example, the origin process may issue numerous AM-based operations, but the target process may be busy with local computation and cannot make progress to complete those operations. On the other hand, currently all metadata for outgoing AM-based operations are managed by the runtime system, an inefficient process because hardware can manage some of them. The runtime system should offload as many operations as it can to the hardware queue and maintain them only when operation metadata exceeds the hardware queue capacity.

Note that some synchronization calls do not require remote completion semantics, such as MPI_WIN_FENCE, MPI_WIN_COMPLETE and MPI_WIN_FLUSH_LOCAL, in which metadata

for remote completion is not needed and the last state can be ignored.

Issuing MPI_ACCUMULATE-Like Operations. In MPI_ACCUMULATE-like operations, when origin data arrives on the target, the runtime system allocates a temporary buffer, places data in that buffer, and then performs computation between the temporary buffer and target buffer. Existing MPI implementations have two scalability problems. One is about derived datatypes. Existing MPI implementations allocate a temporary buffer that has the same datatype and count as does origin data, an approach that wastes a lot of memory if the datatype is sparse. Another problem is that the lack of an upper bound on the size of the temporary buffer because origin data can be extremely large. Open MPI does not support operations with derived datatypes, whereas both MPICH and MVAPICH have both these problems for MPI_ACCUMULATE-like operations.

3.3.5 Consolidating the State-of-the-Art

We consolidate the state-of-the-art of RMA infrastructures from MPICH, Open MPI and MVAPICH into one single baseline implementation: MPI-RMA-base. The final feature set of the consolidated MPI-RMA-base implementation is presented in Table 3.4.

3.4 Design and Implementation of Scalable MPI One-Sided Infrastructure

In this section, we propose a new design of the MPI RMA runtime, called ScalaRMA, to address the scalability challenges discussed in Section 3.3. We describe the design of ScalaRMA from three aspects of MPI RMA: window creation, synchronization, and data movement operations.

Table 3.4: Consolidating the best of the state of the art into MPI-RMA-base

Challenge	**Best of the State-of-the-Art**	**MPICH**	**Open MPI**	**MVAPICH**
Window metadata storage	Baseline memory usage in Table 3.2	✗	✓	✗
FENCE algorithms	RS-based algorithm for AM-based operations and BARRIER-based algorithm for HW-based operations	✓	✓	✓
Metadata for targets	$O(P)$ memory usage	✓	✓	✓
Managing concurrent locks	Queued locks ($O(P)$ memory usage)	✓	✗	✓
Piggybacking or not	Non-piggybacking	✗	✓	✗
Speculative issuing or not	speculative issuing	✓	✗	✓
Operation issuing strategies	Eager issuing for HW-based operations and delayed issuing for AM-based operations	✓	✓	✓
Metadata for outgoing operations	Unlimited memory usage	✓	✓	✓
Issuing MPI_ACCUMULATE-like operations	Unlimited memory usage	✓	✗	✓

3.4.1 Window Creation

As discussed in Section 3.3.2, although we can eliminate $O(P)$ memory usage of window metadata in many situations, in some situations the runtime system still has to consume $O(P)$ memory.

For MPI_WIN_ALLOCATE, we apply "symmetric allocation" on each process so that every process generates the same window base address and does not need to maintain metadata for base addresses anymore. Previous work [27] uses the similar techniques but does not consider shared-memory allocation capability in MPI_WIN_ALLOCATE. In ScalaRMA, we support both symmetric allocation and shared-memory allocation: the root process first decides a base address and broadcasts it to every other processes. Each process then uses that base address to calculate the correct starting address of the shared-memory region and passes it to the

mmap() function.

For the rest of window creation routines, We propose solutions according to different patterns of window information. When the value of window information has certain patterns, for example, the base addresses on all ranks are the same due to symmetric allocation, or window information is set to the same value on all ranks or on odd / even ranks by user, we store the information in a compressed manner. When window information has no patterns, we propose two methods to store the information:

- **Information sharing**: multiple processes share the same window metadata copy. Each process fetches necessary metadata via local load on shared memory or MPI_GET operation on network.

- **Information caching**: each process is attached with a local "cache" which contains the information that is actively used. When the process cannot find information in the current window's cache, it will issue one-sided operations to server processes to fetch the information to the local cache.

In information sharing, depending on how metadata copies are shared among processes, two schemes can be used: (1) fixed number of metadata copies being shared and (2) fixed number of processes sharing the same copy. In the first scheme, the total number of copies is fixed, and therefore the overall memory consumption for the window metadata is constant. However, the communication cost of fetching the metadata is not flat: in the worst case, one process has to talk with P processes in order to fetch metadata. In the second scheme, the memory usage of the metadata copy is not fixed but increases linearly with the number of processes. On the other hand, since the number of processes sharing the same copy is fixed, the communication cost remains constant. Note that in the second scheme the performance can be optimized by configuring the metadata copy in a topology-aware manner, and each process needs to talk only with neighbors within a given physical location. Figure 3.5 compares the differences between these two schemes.

When one process cannot find any metadata in its local memory, it needs to fetch metadata from another process. That process can be on the same node or a different node. If the process is on the same node, the runtime system can place the metadata on a shared-memory region on that node, and the process can fetch it immediately. If that process is on a different node, the origin needs to issue a MPI_GET operation to fetch data, and the performance relies on a fast RMA implementation. The efficiency of fetching metadata can be improved by using a local "cache": when metadata is in a shared-memory region within the same node, that local cache is hardware cache; when metadata is on a different node, ScalaRMA uses a software-based LRU cache on each process to improve accessing efficiency. To guarantee constant memory usage of software-based cache, we use a minimal perfect hashing (MPH) library [28] that can generate a hashing function using the CHD algorithm [29] with memory consumption of 2.07 bits per key. Because MPH requires the set of keys to be static, we need to reconstruct the hashing function whenever a cache miss occurs.

Table 3.5: Comparison of metadata sharing schemes

Metadata sharing scheme	**Memory usage**	**Communication cost**
Fixed number of metadata copies	Constant	In worse case: talks with P processes
Fixed number of processes sharing the same metadata copy	$O(P)$	Talks with fixed number of processes

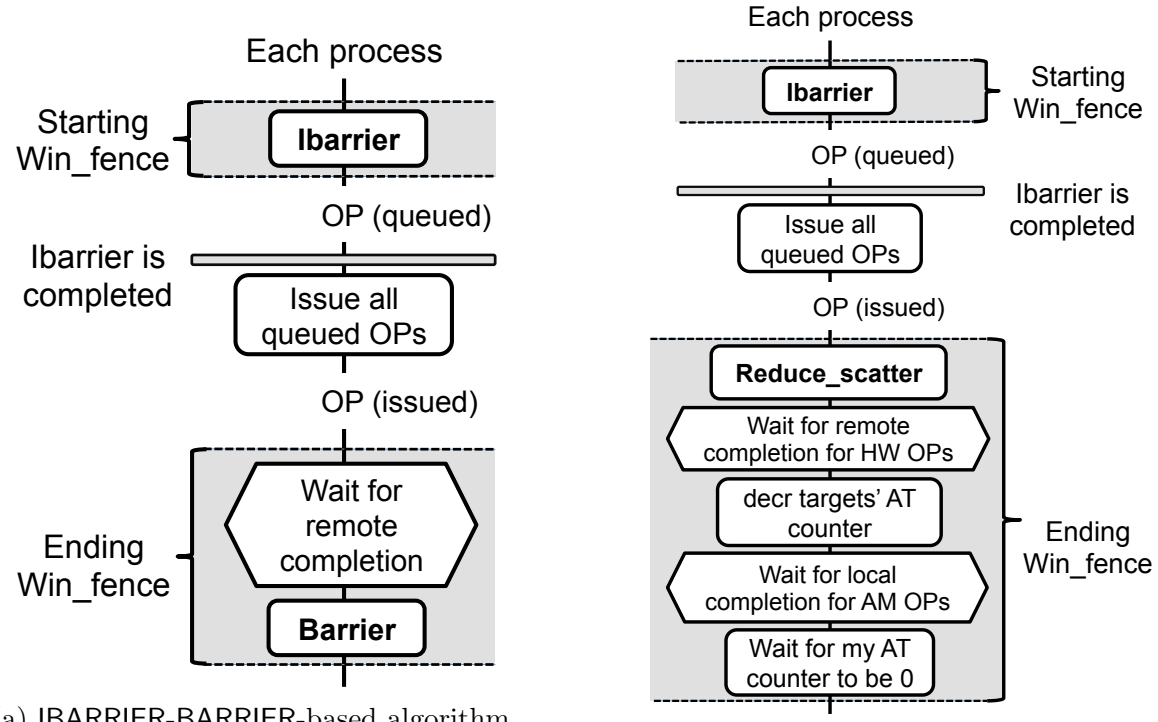
In the Mellanox Infiniband, in order to enable an internode process to share base addresses and scaling unit sizes, every process needs to create an internal window to expose the metadata for remote access, and that window needs to be created by using symmetric allocation to ensure that base addresses are the same and uses constant memory (scaling unit sizes are always the same). Moreover, except for base addresses and scaling unit sizes, the runtime system needs to maintain protection keys for all other processes; and it passes a pointer of each key to the network interface whenever a process invokes an RDMA call. In ScalaRMA, we use internode sharing with local cache: whenever a cache miss happens, the origin process

waits for remote completion of all issued operations to one target before it can replace the MXM key of that target in cache.

3.4.2 RMA Synchronization

In this section, we describe the synchronization algorithm used in ScalaRMA to address scalability and performance challenges.

Active Target Algorithms. In Section 3.3.3, we introduced two FENCE algorithms used in existing MPI runtime systems: a BARRIER-based algorithm and an RS-based algorithm. Here we propose two new algorithms based on them: IBARRIER-BARRIER-based algorithm and IBARRIER-RS-based algorithm. These new algorithms successfully achieve three goals: (1) scalability, (2) issuing of operations as early as possible, and (3) reduced synchronization cost.



(a) IBARRIER-BARRIER-based algorithm

(b) IBARRIER-RS-based algorithm

Figure 3.5: Fence algorithms used in ScalaRMA

In both algorithms, the starting MPI_WIN_FENCE performs an IBARRIER to initialize the synchronization phase and returns immediately. All the following posted operations are queued in the runtime system. As soon as IBARRIER is completed, all queued operations are issued out, and newly posted operations are issued immediately. By using nonblocking communication, the runtime system is able to overlap synchronization with local work.

In the IBARRIER-BARRIER-based algorithm, as shown in Figure 3.5a, the ending MPI_WIN_FENCE is the same as that with the BARRIER-based algorithm in that it first waits for remote completion of all issued operations, then performs a BARRIER, and returns. Its memory usage is flat; however, it introduces more synchronization overhead than what the MPI-3 standard requires. ScalaRMA uses this algorithm for large P. In the IBARRIER-RS-based algorithm, as shown in Figure 3.5b, the ending MPI_WIN_FENCE first performs a REDUCE_SCATTER to determine how many processes have operations targeting itself; it next initializes a local AT counter with that value; then, it waits for remote completion for issued HW-based operations and issues a message to decrement the AT counter on all targets (because targets cannot be aware of completion of HW-based operations but need origins to notify them); then it waits for local completion of issued AM-based operations; and finally it waits for the local AT counter to be zero and return. This algorithm does not have extra synchronization overhead, but the REDUCE_SCATTER call requires $O(P)$ memory usage. ScalaRMA uses this algorithm for small P.

Passive Target Synchronization. In this section, we describe the algorithms used in Passive Target mode, including different algorithms for single lock case and window-wide lock case and scalable algorithm to manage concurrent locks.

In Section 3.3 we noted that each process needs to allocate a target object for each peer in order to maintain the necessary metadata for the passive lock. Those target objects can use $O(P)$ memory in the worst case. In ScalaRMA, for LOCK-UNLOCK synchronization, we still create a new target object whenever MPI_WIN_LOCK is called. That target object

is stored in an internal data structure called the "RMA table," as shown in Figure 3.6. During window creation, the runtime first allocates a slots array with fixed size. Target objects are assigned to corresponding slots in a round-robin fashion. Except for metadata mentioned in Section 3.3.3, the target object also maintains a list of pending operations to the corresponding target.

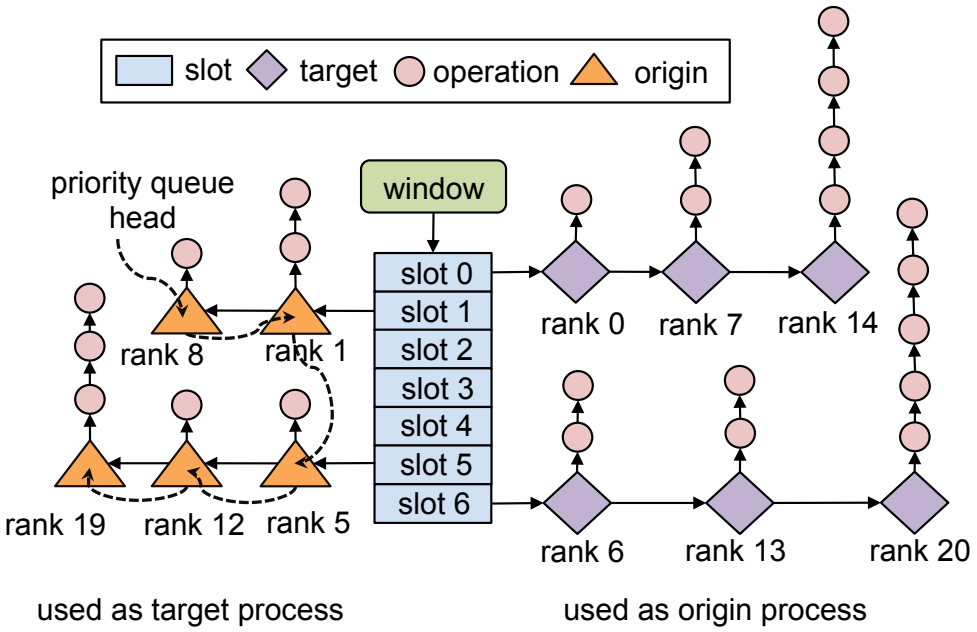


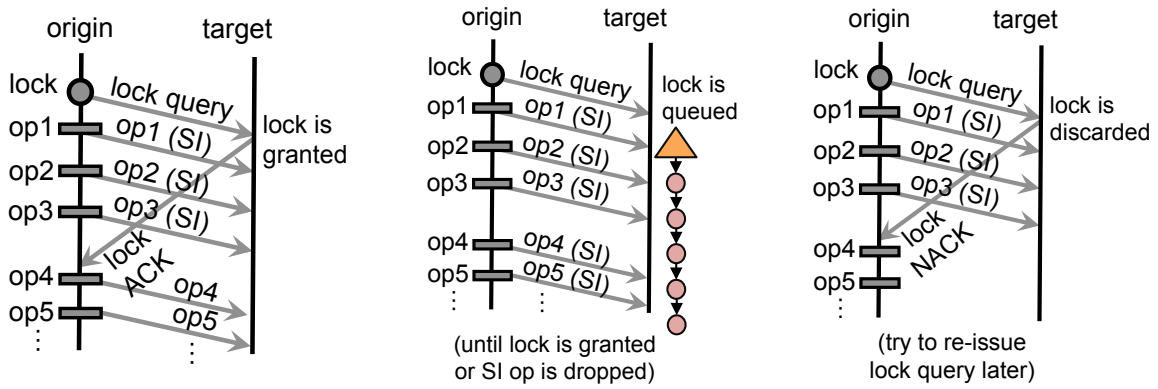
Figure 3.6: RMA table

To avoid creating an unlimited number of target objects, the runtime system preallocates a global pool of target objects sharing among all windows during initialization. When resources in the global pool are used up, the runtime system uses an aggressive clean-up strategy to restore resources: it selects one target object in the RMA table, waits for its synchronization to be finished, flushes out all pending operations in that target object, and waits for remote completion of them. After remote completion is detected, the runtime system can free that target object back to global pool so that the resource is restored. Note that by freeing the target object in the middle of an epoch, we lose metadata for MPI_MODE_NOCHECK and error checking. How to pick a target to restore a resource is critical for performance in aggressive clean-up. If we randomly pick one target, a large number of pending operations may remain

in that target object, and hence aggressive clean-up will be slow. To avoid this situation, we give highest priority for selection to a target that satisfies the following two conditions: (1) pending operations can be issued immediately (synchronization is finished) and (2) the number of pending operations is small. To reduce searching overhead when the number of target objects is large, we use a threshold that specifies a maximum number of targets being visited.

For LOCK_ALL-UNLOCK_ALL synchronization, we want to avoid creating $O(P)$ target objects, and we wish to reduce synchronization cost. We use two protocols to achieve these goals. The first is a "per-target" protocol, which is used when the number of active peers is small. In this protocol, MPI_WIN_LOCK_ALL does nothing and returns immediately; when the origin is going to issue the first operation to one target, the runtime system first issues a lock query and creates a target object for it. In this way the origin creates a target object only for processes it really talks with. When too many target objects are in the runtime system and resources in the global pool are used up, the runtime system uses the second protocol: a "window-wide" protocol. In this protocol, the origin issues a lock query to every process on the window and waits until it receives granted locks from everyone. After that, the origin no longer needs to maintain any target objects.

As with MPICH and MVAPICH, ScalaRMA uses a queued lock strategy to manage concurrent locks instead of network polling, in order to avoid expensive network traffic. However, a memory management strategy is required to prevent a lock query queue on the target from using up internal resources. In ScalaRMA, we use an "origin object" to store information about the lock query from each origin, as shown in Figure 3.6. As with the target object, we provide a global pool to supply the resources of origin objects on the origin process. When a lock query arrives, the target fetches one origin object from the global pool and enqueues it into both the priority queue and the corresponding slot. When origin objects are used up, the target discards the current lock query and sends a notification to the corresponding origin; the origin then tries to issue a lock query again later.



(a) Lock query is granted (b) Lock query is queued (c) Lock query is discarded

Figure 3.7: Speculative issuing strategy

In Section 3.3.3 we noted that MPICH and Open MPI use two optimization: piggybacking locks (PB) and speculative issuing (SI). PB can save network traffic by reducing the number of messages; however, the benefit becomes smaller as the communication volume increases, and it delays issuing of a lock query until the first operation is posted, thus losing the opportunity of overlapping communication with local computation. In ScalaRMA, we drop the PB strategy because we want to issue a lock query as early as possible; and we keep the SI strategy in order to issue as many operations as possible before a lock is granted, until the origin receives notification of dropping operations from the target. Figure 3.7 illustrates this model in various cases.

When an SI operation arrives on the target, the target needs to decide whether it can perform that operation (previous lock query is granted), queue that operation (previous lock is queued), or discard that operation (previous lock is discarded). The target makes such a decision by searching for a corresponding origin object in the RMA table. For shared locks, the target needs to maintain an origin object in the RMA table, not only for queued locks, but also for granted locks. For exclusive locks, the target needs to maintain an origin object in the RMA table only for queued locks. We note that maintaining an origin object for a granted shared lock is needed only when shared locks have been discarded. Therefore, we optimize the memory usage of the origin objects by keeping track of the number of shared

locks being dropped and not retried and by creating an origin object for a granted shared lock only when it is not zero. When a lock is discarded, notification from the target to the origin will disable the SI strategy on the origin side because all future SI operations from that origin will be discarded as well. Note that even though we implement such SI strategy only in PT mode, it can be extended to AT mode as well to speculatively issue operations before IBARRIER synchronization is finished.

3.4.3 Data Movement Operations

In this section, we present scalable solutions to issue data movement operations in MPI RMA infrastructure, including both HW-based operations and AM-based operations.

Multi-Layered Design to Support Different RMA Implementations. Our framework supports both AM-based operations and HW-based operations in MPI runtime using a multi-layered design. MPI runtime provides corresponding function pointers for each RMA operation, and the network module overwrites those function pointers when it can implement those operations. The AM-based implementation in software serves as a fallback when the network module does not provide the implementation. As is indicated in Figure 3.8, when the RMA operation is invoked in the application, the runtime first invokes the corresponding implementation in network module; if the hardware implementation is not available, a software implementation is triggered as a fallback, which is using the SEND / RECV operations in network module to implement the one-sided capability. In such case the target side keeps receiving RMA messages in the progress engine by making MPI calls.

Issuing with "Bounded Laziness". In Section 3.3.4 we introduced two issuing strategies used in MPI implementations: eager issuing and lazy issuing [30]. Here we discuss in detail about how each of them works.

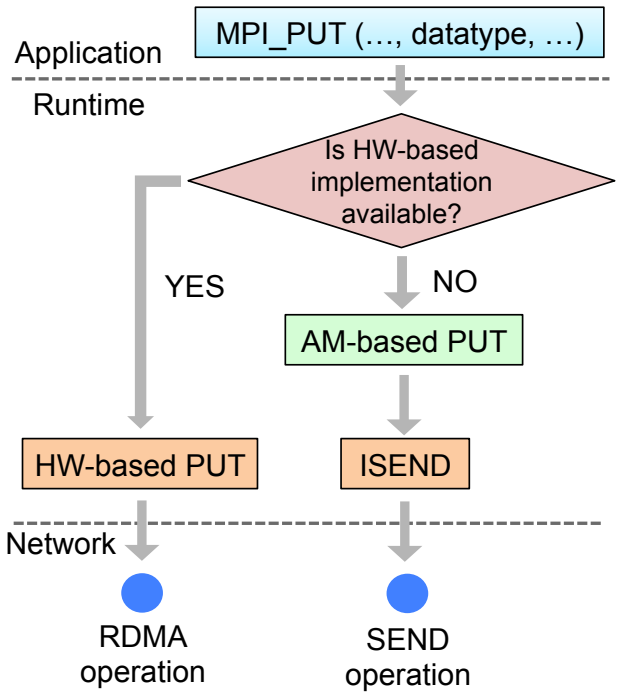


Figure 3.8: Multi-layered Design for MPI RMA operations

- **"Eager" strategy**: in the synchronization call that opens an epoch, origin process first performs proper synchronization with target processes; and all following RMA operations are issued immediately after they are posted; in the synchronization call that ends an epoch, origin process performs proper synchronization with targets again to guarantee that all previous RMA operations are completed. It is a native choice of implementation, which involves two synchronization for each epoch: one is at beginning call and another is at ending call. Every RMA operation is issued in an "eager" pattern, namely, as soon as being posted.

- **"Lazy" (delayed) strategy**: in the synchronization call that opens an epoch, origin process and target process do nothing; all following posted RMA operations are queued internally in MPI runtime; in the synchronization call that ends an epoch, origin process does proper synchronization with targets and then issues out all queued operations; finally origin process merges the ending synchronization with the last operation sent to each target.

Currently most MPI implementations implement MPI RMA interface using lazy strategy, because it has the advantage of lower synchronization overhead when number of operations is small. However, for the irregular application which involves large number of outgoing operations, lazy strategy hurts the scalability and can potentially cause the application to be aborted. The reason is that, in order to delay issuing operations to the end, the MPI runtime has to internally queue up all posted operations and maintain the states of them.

In ScalaRMA, we propose a new issuing strategy, called "bounded laziness" issuing, to combine the advantages of those two strategies. In bounded laziness issuing, we perform nonblocking communication at the beginning of an epoch—IBARRIER in FENCE (Figure 3.5b), ISENDs / IRECVs in PSCW, LOCK-UNLOCK, and LOCK_ALL-UNLOCK_ALL—to initialize the synchronization phase. All epoch-opening calls are nonblocking. Issuing parameters of the following posted operations are stored in the operation object and queued in the corresponding target object, as shown in Figure 3.6. The progress engine on each process keeps checking whether the synchronization phase is completed. As soon as it is finished, the runtime system issues all queued operations; and all newly posted operations are issued out immediately without queuing. A blocking synchronization is performed at the end. By using nonblocking communication at the beginning, we overlap synchronization with local work on each processes.

The strategy of bounded laziness can be considered as a hybrid strategy which combines advantages of both eager and lazy strategies: in the synchronization call that opens an epoch, origin processes and target processes do synchronization using nonblocking operations, and return without waiting for its completion; when an operation is posted, the runtime will check if the synchronization is completed or not, if completed, it will issue all operations currently in the system (queued and currently posted), otherwise it will queued the current operation; in the synchronization call that ending an epoch, origin processes does another synchronization with the target processes.

A naive implementation for MPI_WIN_FENCE is: each process invokes a BARRIER in the opening fence, then issues operations eagerly and waits for the remote completion for all operations, and finally invokes another BARRIER in the ending fence. Even though this implementation guarantees the correctness of MPI_WIN_FENCE, the acknowledgment of remote completion brings significant overhead. Another implementation option is that, each process still issues operation eagerly, but just detects the local completion for operations, after that, each process invokes collective communication (REDUCE_SCATTER) so that every process can know how many operations targeting at me, and it set a local counter to that value and waits until receiving such number of operations. The second option eliminate the overhead of acknowledgment messages, however, the RS-based algorithm requires an $O(P)$ data structure. Therefore, in our work, we use RS-based algorithm for small P and switch fence to BARRIER-based algorithm when P is large.

Before nonblocking communication is completed, there is still operation metadata (issuing parameters) buffered in the runtime system that has the risk of using up internal resources. We use an aggressive clean-up strategy similar to that used with target objects to restore operation objects. The only difference is that for operation objects, we do not need to wait for local or remote completion, because operation objects are no longer needed after an operation is fully issued out. Apart from operation objects, there are also request objects in the runtime system that keep track of issued but incomplete operations. To prevent them from using up memory, we set the maximum number of request objects that can exist in the runtime system; if the actual number exceeds that level, the runtime system will wait until the number of request objects is reduced.

Issuing MPI_ACCUMULATE-Like Operations. In ScalaRMA, we solve the problem of MPI_ACCUMULATE-like operations by first staging derived datatype data from origin in a packed buffer on target, so that there is no waste space on temporary buffer, and secondly streaming a large MPI_ACCUMULATE-like operation into multiple small units,

so that the size of temporary buffer is fixed on the target side and communication and computation can be pipelined.

3.4.4 Resource Management Strategies

In an MPI implementation using delayed strategy, the internal resources are very likely to be used up by MPI runtime when the application involves large number of RMA operations. The reason is that, the runtime has to internally maintain all operations posted within the epoch and delay issuing them out till the ending synchronization call. Due to this reason, our implementation issues out operations as early as possible. However, the runtime still needs to maintain some RMA operations internally in the following two cases: (1) when the RMA synchronization has not been completed, the runtime cannot issue posted operations but must internally maintain them; (2) if one operation is issued but is not completed immediately, the runtime needs to maintain that operation until it is completed. The data structure used to maintain such operations is significant for performance. One option is to store operations separately according to the target rank. This allows further optimization on operations with the same target rank, but it introduces an unscalable $O(P)$ data structure. Another option is to store all operations in a single linked list. It does not have scalability problem but prohibits any further optimization related to the same target ranks.

Our implementation uses a three-dimensional data structure that combines the characteristics of the above two methods. The data structure, called RMA operation table, is illustrated in Figure 3.6. When creating the RMA window, a fixed size of array is allocated and attached to the window. Each slot in the array maintains a list of states of current active targets. Each target contains a list of pending operations, a list of issued incomplete operations, and other important states like synchronization states.

The RMA progress engine on the origin side includes two types of work: completing RMA synchronization, and issuing RMA operation (communication). We first check if RMA synchronization is completed, if so, we issue out RMA operations as many as possible;

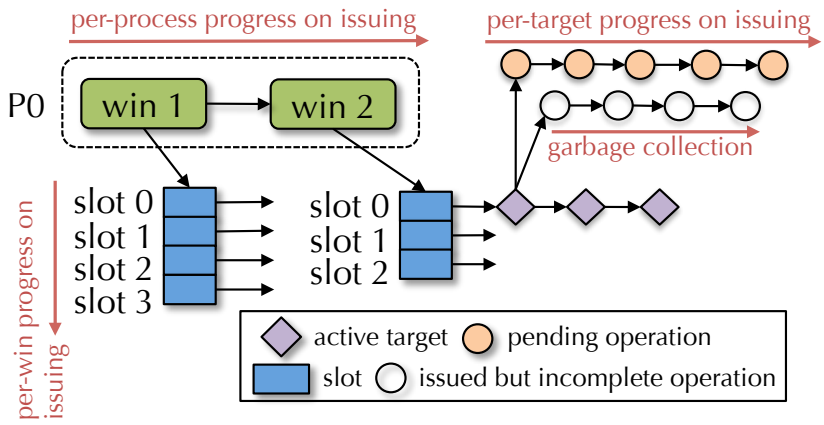


Figure 3.9: Multi-level strategy for making RMA progress

otherwise we exit the progress engine.

On the origin side, we use a multi-level strategy to trigger the RMA progress engine. The first and the most general level is *per-process*: the progress engine makes progress on all windows on the origin process. This is called from the normal progress engine of MPI; the second one is per-window: the progress engine makes progress on one specific window. It is typically used in the ending synchronization calls with window argument (MPI_WIN_FENCE, MPI_WIN_COMPLETE, MPI_WIN_FLUSH_ALL), since they care about the completion of all operations on the current window; the third one is per-target: the runtime makes progress on operations to one specific target. This one is used in RMA communication calls and synchronization calls for single target (MPI_WIN_UNLOCK and MPI_WIN_FLUSH).

In addition to making progress on pending operation list, we also need to make progress on issued operation list. The operations in issued list are ones that are issued out but not completed yet. If the runtime discover operations in the issued list that are completed now, it can be removed from the issued list and free.

3.4.5 Making Efficient RMA Progress

To achieve efficient RMA progress, we define two types of RMA windows in ScalaRMA:

- **Inactive window**: where a process does not have work to do in this window unless there are either user events or network events happening (e.g., an operation is posted by the user or synchronization is finished by the network).
- **Active window**: where a process has work to do on this window even though neither a user event nor a network event is happening. By identifying active and inactive windows, the runtime system can know when to execute RMA progress and avoid unnecessary progress execution.

In the runtime system we maintain a list for both RMA window types: "active window list" and "inactive window list". Every newly created window is first added to the inactive window list. When we set one window to active, we move it from the inactive window list to the active window list. When we add the first window to the active window list, we activate the RMA progress. When we remove the last window from the active window list, we deactivate the RMA progress. By doing so, the RMA progress will not be triggered unless there are active RMA windows in the runtime system.

3.4.6 Trade-offs Between Scalability and Performance

In ScalaRMA, there are several places that we need to trade off between scalability and performance. When MPI runtime provides more memory, the runtime will spend less communication cost. If we want to achieve constant memory consumption, spending more communication cost is inevitable. Table 3.6 lists all trade-offs involved in the design of ScalaRMA. We use communication-efficient algorithms when P is small, but switch to memory-efficient algorithms when P is large.

3.5 Experimental Evaluation

We used two clusters for our evaluation. The first is configured with Mellanox InfiniBand and has 320 nodes (each node has 8 cores). The second consists of 16 nodes (each node has

Table 3.6: Trade-offs between scalability and performance in ScalaRMA

Aspects	**Comparison**	**Memory usage**	**Communication cost**
Window metadata storage	Fixed number of metadata copies	$O(1)$	In worse case: talks with P processes
	Fixed number of processes sharing the same metadata copy	$O(P)$	Talks with fixed number of processes
Data movement operation	HW-based operation	$O(1)$	More cost: waits for remote completion in AT mode
	AM-based operation	Relative to problem size	Less cost: waits for local completion in AT mode
FENCE algorithms	IBARRIER-BARRIER-based algorithm	$O(1)$	More cost: waits for remote completion in AT mode
	IBARRIER-RS-based algorithm	$O(P)$	Less cost: waits for local completion in AT mode
LOCK-UNLOCK algorithms	No lock query pool	In the worse case $O(P)$	Less cost: origin does not need to re-issue lock query
	Lock query pool	$O(1)$	More cost: Origin may need to re-issue lock query
LOCK_ALL-UNLOCK_ALL algorithms	Per-target alforithm	In the worse case $O(P)$	Less cost: origin issues lock query only to targets it really talks with
	Window-wide algorithm	$O(1)$	More cost: origin issues lock query to all P processes

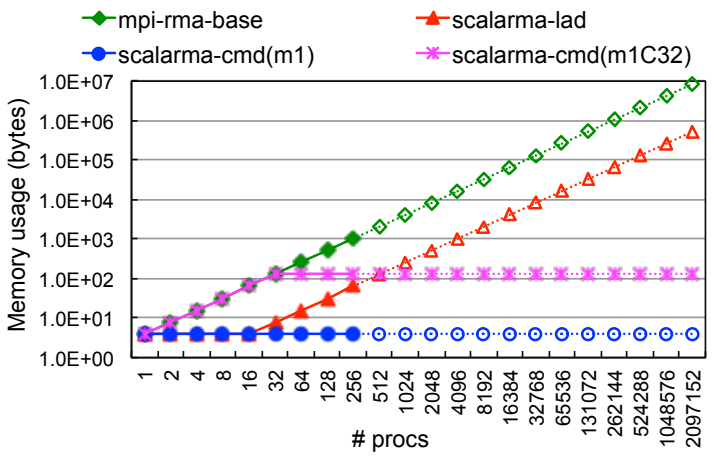
16 cores) configured with the Portals-4 network. MPI_PUT and MPI_GET with basic datatypes are implemented in hardware, and the rest of operations are AM-based implementation.

3.5.1 Microbenchmarks

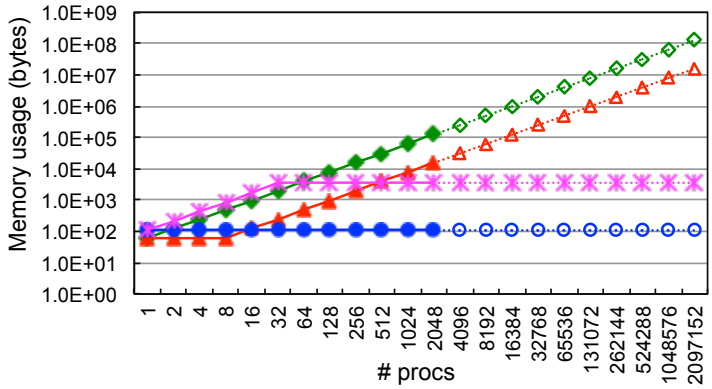
In this section, we present micro-benchmark results of ScalaRMA, from perspective of window creation, synchronization algorithms and data movement operations.

Window Creation. In this section we measure the impact of different window metadata schemes on memory usage and communication. In our tests, every process creates a window with a different scaling unit size using MPI_WIN_CREATE and performs all-to-all communication using MPI_PUT. Operations are issued in batches of 10. We compare MPI-RMA-base with three schemes in ScalaRMA. The first is locality-aware distribution ("ScalaRMA-LAD"), in which a fixed number of processes share the same copy; it is implemented by intranode sharing via a shared-memory region. The second scheme is constant memory distribution ("ScalaRMA-CMD"), which is implemented by internode sharing; each process keeps only its own window metadata and issues a MPI_GET to fetch metadata. The third is a combination of the second scheme and a software-based cache of 32 cache lines ("ScalaRMA-CMD-cache").

Figure 3.10a shows memory usage on the Portals-4 network, in which the window metadata to be maintained by the runtime system includes the scaling unit size (4 bytes). The results are gathered from 1 process to 256 processes, and dot lines after 256 processes are the trend lines based on current results. In MPI-RMA-base, $O(P)$ memory is used to store window metadata for all processes. In ScalaRMA-LAD, 16 processes within one node share the same copy, and therefore memory usage on each process is largely reduced; however, it is still increased with P and not scalable. In both ScalaRMA-CMD and ScalaRMA-CMD-cache, as $O(P)$ grows, the maximum number of metadata copies is fixed, and memory usage is reduced to $O(1)$.



(a) Memory usage (Portals-4)

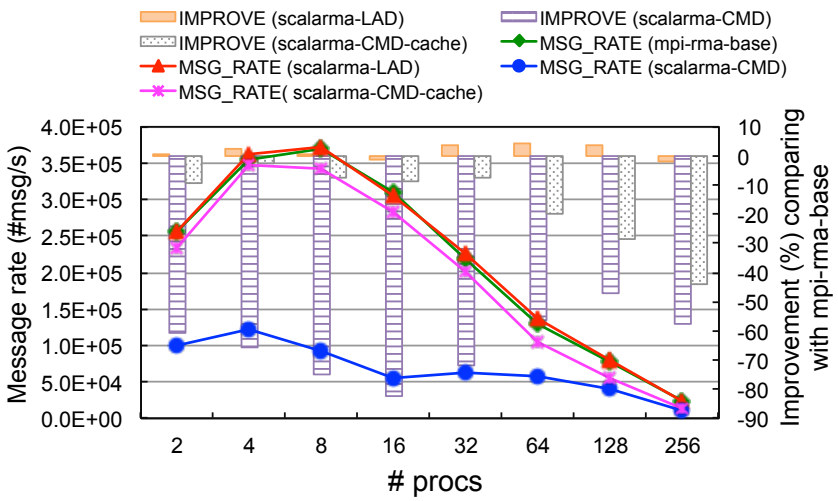


(b) Memory usage (MXM)

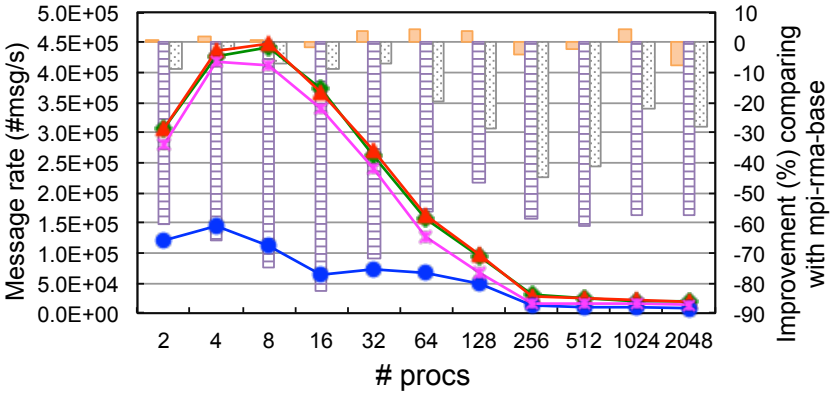
Figure 3.10: Memory usage of different window metadata schemes in MPI_WIN_CREATE

On Sequoia machine, totally there are 1,572,864 cores. If we run one process per core with MPI-RMA-base, each process needs to consume 6.3MB per window and each node needs to consume 100MB memory per window. Suppose an application generates 64 windows at one time, each node needs to consume 6.4GB for window metadata. This already uses up 40% of memory on each node on Sequoia.

Figure 3.11a shows the message rate on the Portals-4 network. Both MPI-RMA-base and ScalaRMA-LAD can achieve the best performance because they can directly access the window metadata stored on the local process or local node. ScalaRMA-CMD is about 80% worse than MPI-RMA-base, however, because each process needs to do an additional



(a) Message rate (Portals-4)

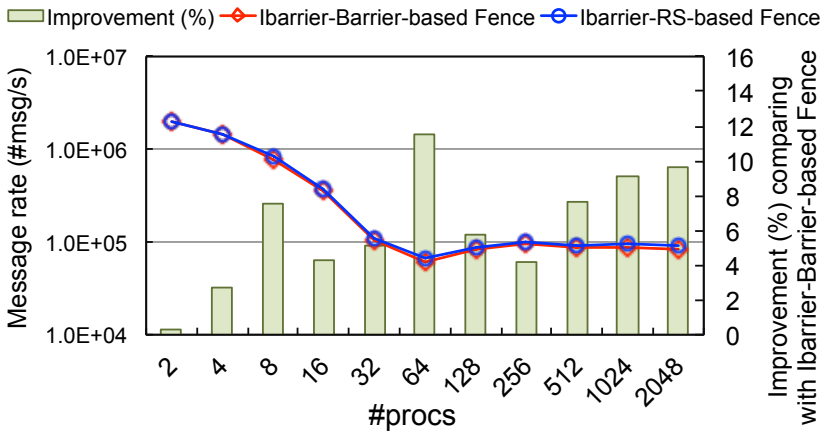


(b) Message rate (MXM)

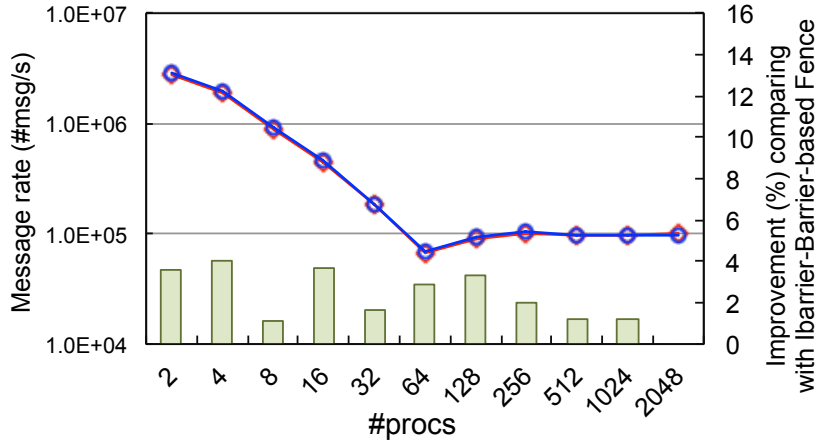
Figure 3.11: Message rate of different window metadata schemes in MPI_WIN_CREATE

MPI_GET before each RMA operation. ScalaRMA-CMD-cache optimizes ScalaRMA-CMD by maintaining metadata in local cache; however, it still has some performance loss compared with MPI-RMA-base because of cache misses, in which the runtime system needs to reconstruct the MPH function.

Figure 3.10b shows memory usage on the MXM network, in which window metadata that needs to be maintained by the runtime system includes the scaling unit size (4 bytes), base address (8 bytes), and MXM key (48 bytes). In MPI-RMA-base and ScalaRMA-LAD, the memory consumed is relative with number of processes and keeps increasing when number of



(a) AM-based operations only



(b) Mix of AM-based and HW-based operations

Figure 3.12: Comparison between two FENCE algorithms in ScalaRMA (MXM)

processes is increased. In ScalaRMA-CMD and ScalaRMA-CMD-cache, the memory usage keeps constant. Note that the metadata maintained for each rank in ScalaRMA-CMD and ScalaRMA-CMD-cache is slightly larger than MPI-RMA-base and ScalaRMA-LAD because we need to maintain two MXM remote keys, one for user window and one for internal window (used for inter-node sharing). Size of metadata maintained for each rank in Mellanox InfiniBand is much larger than the size in Portals-4, therefore it is more easily to use up internal resources on large scale machine like Sequoia. Figure 3.11b shows the message rate on MXM, which has a trend similar to that seen with the Portals-4 results.

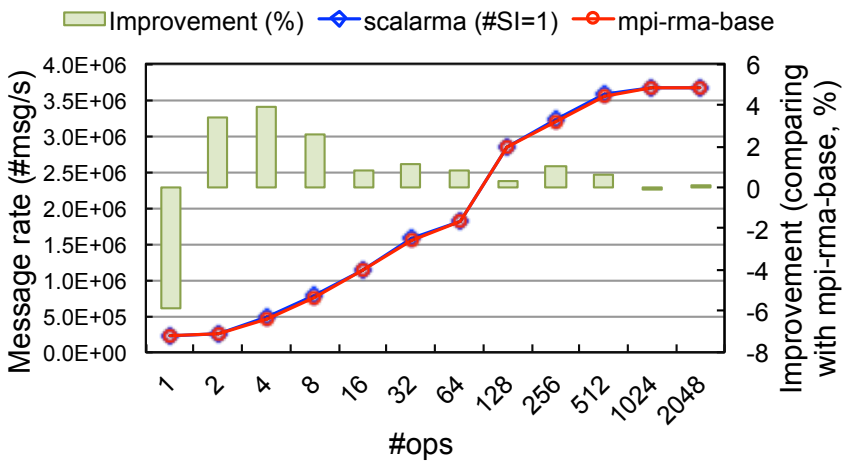
Synchronization.

In this section, we present experimental results for synchronization algorithms, including FENCE synchronization and passive lock synchronization algorithms.

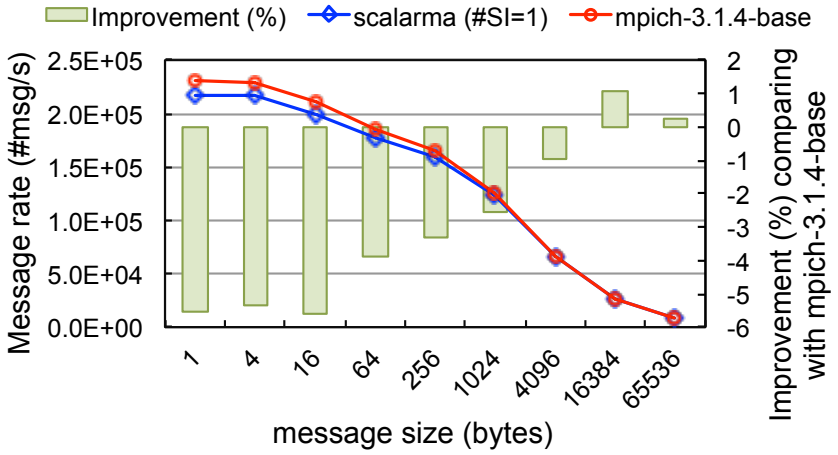
Here we compare the performance of two FENCE algorithms in ScalaRMA: the IBARRIER-RS-based algorithm and the IBARRIER-BARRIER-based algorithm. The microbenchmark performs an all-to-all communication, in which each process issues 1,000 small RMA operations to everyone else. Figure 3.12a shows the performance with AM-based operations only. The performance of the IBARRIER-RS-based algorithm is about 12% better than that of the IBARRIER-BARRIER-based algorithm because the IBARRIER-RS-based algorithm waits for local completion, whereas the IBARRIER-BARRIER-based algorithm waits for remote completion. Figure 3.12b shows the performance of a mix of AM-based and HW-based operations. The gap between the two algorithms is reduced because the IBARRIER-RS-based algorithm has to wait for remote completion for HW-based operations. If a network can perform remote notification, its performance can be improved.

Here we first compare MPI-RMA-base, which has a piggybacking (PB) strategy, with ScalaRMA, which does not have this strategy but speculatively issues one operation. As shown in Figure 3.13a, the performance of MPI-RMA-base is about 6% better than that of ScalaRMA when there is only one operation between MPI_WIN_LOCK and MPI_WIN_UNLOCK and the message size is below 16K bytes. As the number of operations grows or the message size increases, ScalaRMA becomes better than or similar to MPI-RMA-base. Many applications involve massive outstanding messages within an epoch, in which case the benefit from PB to those applications is limited.

In Figure 3.14 we evaluate the impact of different numbers of speculatively issued (SI) operations in ScalaRMA. In the test, the origin issues a shared lock to the target, and that lock is granted immediately. We observe that as we increase the number of SI operations, we can fill more operations issuing with a network waiting time of the lock acknowledgment. After 700 SI operations, performance no longer increases; that is, the issuing of SI operations



(a) Piggybacking vs. non-piggybacking for multiple operations



(b) Piggybacking vs. non-piggybacking for single operation

Figure 3.13: Performance of piggybacking and speculative issuing strategies (MXM)

is fully overlapped with network transaction of the lock acknowledgment.

3.5.2 Data Movement Operations

In this test we compare bounded laziness issuing in ScalaRMA with delayed issuing in MPI-RMA-base. Specifically, we measure the message rate of all-to-all communication by varying three metrics: message size, number of operations, and number of processes. Figure 3.15 compares ScalaRMA with MPI-RMA-base. When there is a single operation, the message size is smaller than 64 bytes, and the number of processes is only 2, ScalaRMA is worse

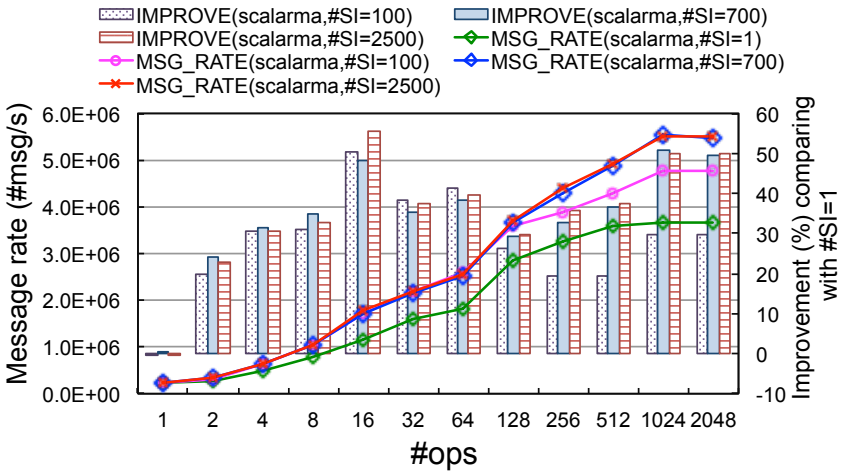


Figure 3.14: Impact of speculatively issued operations

than MPI-RMA-base. As the message size, number of operations, and number of processes increase, however, ScalaRMA performs the same as or better than MPI-RMA-base. When the number of operations is 4,096, ScalaRMA is better than MPI-RMA-base—around 10% at maximum.

3.5.3 Evaluation with Mini-apps

In this section, we present experimental results evaluating with mini-apps, including Graph 500 benchmark. Graph 500 is a supercomputing benchmark used to test data intensiveness for runtime systems and large-scale systems. It generates a graph with vertexes and edges and performs a breadth-first search on that graph; its performance metric is traversed edges per second. The MPI RMA implementation of Graph 500 performs a large number of small MPI_ACCUMULATE operations among all the peers during FENCE epochs. We ran the Graph 500 benchmark with both ScalaRMA and MPI-RMA-base with strong scaling and weak scaling. The results are shown in Figure 3.16a and Figure 3.16b. In both cases, MPI-RMA-base fails to finish the benchmark when the problem size on each process is large, because the operation metadata maintained in the runtime uses up all internal resources. On the other hand, ScalaRMA can finish running the benchmark for any problem size, because the resource management strategy in ScalaRMA guarantees that memory consumption by the

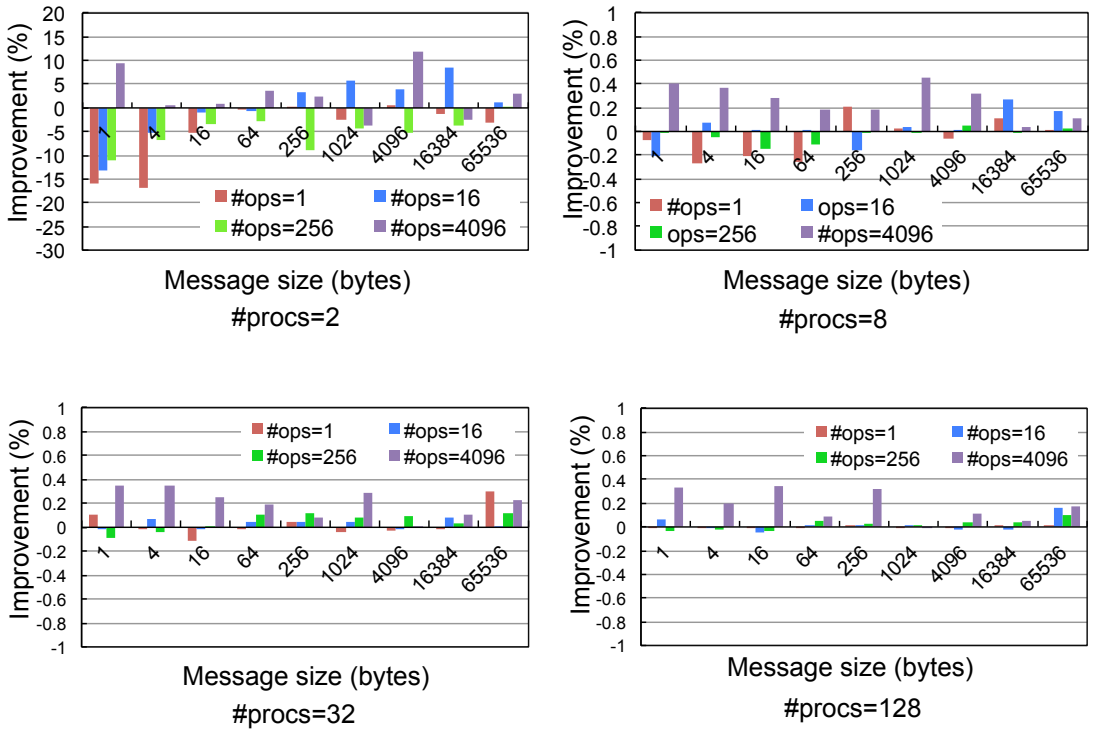


Figure 3.15: Message rate with increasing message size, number of operations and number of processes (MXM)

runtime system is restricted within a certain amount.

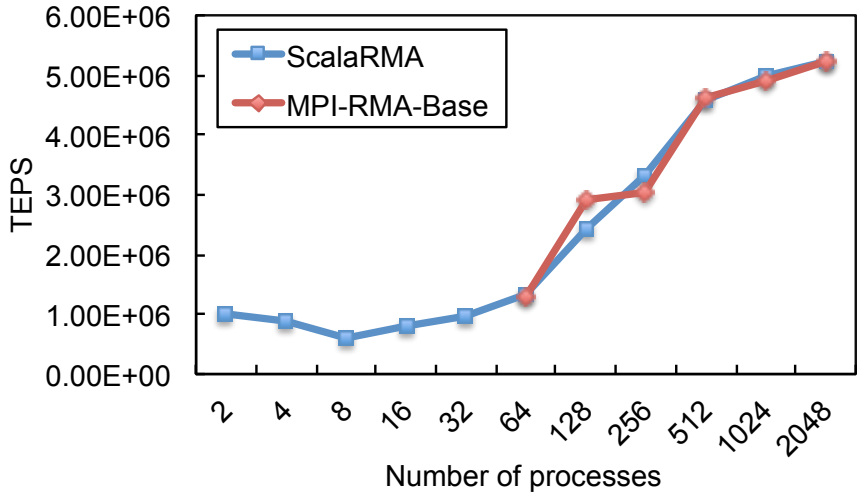
3.6 Related Work

Several researchers have studied the scalability of MPI implementations with respect not only to MPI one-sided communication but also to other aspects in the MPI library. In [31], Balaji et al. address several significant issues related to scaling MPI to millions of cores, in terms of what is needed both in the MPI-3 specification and in MPI implementations. In [27], Gerstenberger et al. present a design and implementation of MPI-3 over the Cray Gemini and Aries systems; the design directly uses RDMA features from hardware to implement scalable protocols for MPI RMA synchronization. In [32] [33] [34], the authors propose memory-space-related optimization for MPI implementations, such as the memory required for storing communicators and groups. The operation issuing strategy is studied in [26], in which REDUCE_SCATTER-based FENCE algorithm and delayed issuing are proposed in order to

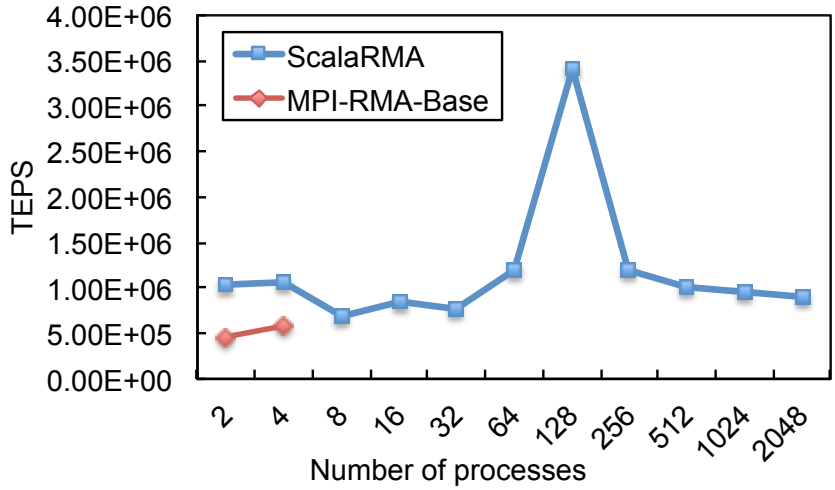
reduce synchronization overhead over TCP networks. In [35], an issuing strategy is proposed that can adaptively switch from delayed issuing to eager issuing in an MPI runtime system. The implementation of MPI-2.2 RMA over InfiniBand has been discussed in [36] [37] [38]. In [39], Zounmevo et al. discuss a scalable message queue mechanism for large-scale jobs. In [40], Darius et al. implement MPI runtime over the Nemesis communication system and streamline the critical path to reduce the overhead of sending and receiving a message for intra-node communication. Even though the work effectively reduces the instructions over MPI stack, there is still more overhead in MPI runtime compared to other programming models such as SHMEM and UPC. To the best of our knowledge, no previous work has addressed all the scalability limitations in current MPI one-sided implementations.

3.7 Conclusion

In this section, we propose a new MPI RMA runtime system, called ScalaRMA, to address scalability challenges in MPI one-sided communication. Existing MPI implementations have several limitations in MPI RMA with respect to window creation, synchronization, and data movement operations. These limitations prevent the MPI runtime system from successfully scaling applications onto large-scale machines, because metadata for processes and outstanding operations uses up all internal resources. ScalaRMA provides new algorithms and strategies to manage different kinds of metadata in MPI one-sided communication on different hardware interconnects. Approximately 80% this work is included in MPICH release with production quality.



(a) Strong-scaling Graph 500 with problem size of 2^{22} vertices



(b) Weak-scaling Graph 500 with problem size of 2^{19} to 2^{29} vertices

Figure 3.16: Experimental results of Graph 500

CHAPTER 4

Adaptive Issuing Strategy for MPI One-Sided Communication

There are two issuing strategies commonly used in current popular MPI one-sided implementations: the first strategy is called "eager issuing" approach, in which the epoch-opening RMA synchronization call performs a blocking synchronization; all following operations are issued as early as possible; the epoch-closing RMA synchronization call performs another blocking synchronization call at end. Another strategy is called "delayed issuing" approach, in which the epoch-opening RMA synchronization call does nothing and returns immediately; all following operations are queued up and issued in the epoch-closing RMA synchronization. By doing so, the RMA synchronization overhead at beginning can be eliminated. Delayed issuing approach has advantages for communication with small data because of the reduced network operations, whereas eager issuing approach can achieve better performance for large data transfers. In this chapter, we describe a design and implementation of an adaptive issuing strategy for MPI one-sided communication, which can be used with various MPI RMA synchronization modes, including FENCE, POST-START-COMPLETE-WAIT (PSCW) and LOCK-UNLOCK. Such adaptive strategy can combine the benefits from both delayed issuing and eager issuing approaches. We conduct a thorough performance evaluation and the experimental results show that the adaptive approach performs as well as the delayed issuing approach when data volume is small and can achieve similar performance with eager issu-

ing approach when data volume is large. Apart from this, our approach can achieve good overlapping performance of communication and computation.

4.1 Overview

The MPI-3.0 specification provides a lot of flexibility on when an MPI one-sided operation can complete, which allows an MPI one-sided implementation to be optimized internally, specifically in terms of when data transmission are initialized within MPI RMA epoch.

When number of operations is small and data transferred in each operation is short, issuing those operations at the ending of the epoch allows the aggregation of the operations into fewer communication operations, which can improve the performance. On the other hand, when number of operations is large or data transferred is large amount, issuing those operations as early as possible may be beneficial, this is because their transmission latency is much more expensive and issuing them early enough allows the opportunities to overlap the communication with local computation.

In many situations in MPI application, it is not clear beforehand that when eager issuing is better and when delayed issuing is better, since the communication pattern tends to be changed throughout the execution. Therefore, it is necessary for MPI runtime to have an adaptive issuing strategy, which can automatically select the suitable issuing strategy based on the current situation in MPI application. Such adaptive strategy takes over the burden from the user to understand MPI implementation choices in detail, and avoid the possibility of making mistakes by the user, since MPI runtime can adaptively select the most suitable approach. In this work, we design and implement an adaptive issuing approach for MPI one-sided communication.

4.2 Adaptive Strategy Design

In this section, we describe the adaptive algorithm for different three MPI RMA synchronization: LOCK-UNLOCK, POST-START-COMPLETE-WAIT, and FENCE synchronization.

4.2.1 LOCK-UNLOCK Synchronization

The existing implementation of LOCK-UNLOCK in MPICH uses a "delayed issuing" strategy. In beginning MPI_WIN_LOCK, the origin process does nothing but enqueues the lock request and returns immediately. During the following synchronization epoch, the origin process enqueues all posted MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs. In the ending MPI_WIN_UNLOCK, the origin process first issues out the lock request and waits for the lock granted acknowledgment from the target process. After that, it issues all queued operations and sets a field in the packet header in the last operation to make the target process decrements the counter at the window. When that counter becomes zero, all operations have already arrived at the target side. By doing so, the delayed issuing approach is able to combine the synchronization message with the last data movement operation. The delayed approach also contains an optimization for single small data movement operation: if there is only one operation between MPI_WIN_LOCK and MPI_WIN_UNLOCK, the data size is small and the MPI datatype is predefined, the origin process sends that operation together with the beginning lock request in MPI_WIN_UNLOCK. In such situation, both synchronizations at the epoch-opening and at the epoch-closing can be eliminated. Another choice for LOCK-UNLOCK is an "eager issuing" approach. In this approach, during MPI_WIN_LOCK, the origin process issues the lock request as soon as possible and waits for lock to be granted before it returns. For all following posted MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs, it issues them as soon as they are posted in the application. In the ending MPI_WIN_UNLOCK, the origin process issues a 0-byte message to release the lock on the target. Such eager issuing approach needs two synchronizations operations and three messages: one message for the

lock request and one for the lock acknowledgment, and one message for the unlock request at the end. On the other hand, since it issues the lock request and operation separately instead of merging them together, the optimization for single short operation described above is impossible in eager issuing strategy. Our adaptive design for LOCK-UNLOCK eliminates the synchronization message at the epoch-closing and still keeps the optimization for single short operation. In MPI_WIN_LOCK, the origin process enqueues the lock request and returns immediately, just like what delayed issuing does. All following posted RMA operations are enqueued in the runtime. If the number of queued operations reaches a certain threshold, which is the value of operation number or message size, the origin issues the lock request immediately, but does not wait for the lock granted acknowledgment. Instead, it continues to queue up RMA operations until the lock is granted. Once the lock acknowledgment is back, it issues all queued operations, switches from delayed issuing to eager issuing, and issues all the following operations as early as possible. Even though the rest of operations are issued in an eager manner, we still can avoid the synchronization in the epoch-closing by maintaining a pointer to the current last RMA operation, which keeps one operation not issued until the ending MPI_WIN_UNLOCK. We also preserve the optimization of single short operation in the new design, since the lock request is not issued in MPI_WIN_LOCK, if there is only single short operation existing within the epoch, it will be issued together with the lock request in MPI_WIN_UNLOCK. The semantic of MPI_WIN_UNLOCK in MPI standard requires that when the function returns, MPI one-sided operations are completed at both origin side and target side. We use an optimization strategy to ensure this. For shared lock, when origin meets a MPI_GET operation, the origin keeps it in a buffer and issues it at last, otherwise the target needs to explicitly issue an acknowledgment message to the origin after receiving the last operation. This strategy assumes that the underlying network is ordered. If the underlying network hardware is unordered, the acknowledgment message is always needed at last. For exclusive lock, no acknowledgment is needed since there will be only one origin modifying the memory region on target process.

4.2.2 POST-START-COMPLETE-WAIT (PSCW) Synchronization

The current implementation of PSCW in MPICH also uses a delayed issuing strategy. In beginning MPI_WIN_POST, processes in the target group issues a synchronization message to each process in the origin group, and sets the initial counter on the window to the size of the origin group. In beginning MPI_WIN_START, processes in the origin group do nothing and return immediately. The following posted MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs are all queued in the MPI runtime. In ending MPI_WIN_COMPLETE, every process in the origin group is first blocked until it gets all synchronization messages from all processes in the target group, after that, it issues all queued operations. For target process of each operation, the origin process sets a field in the packet header of the last operation in order to decrement the counter on the window. In MPI_WIN_WAIT, every process in the target group is blocked until its own counter on the window becomes zero. For each pair of processes between origin group and target group, only one synchronization message is needed. If the origin process has no operation issued to one target process, it needs to explicitly issue an additional 0-byte message to that target process, which means that they need two synchronization messages. Another choice for the implementation of PSCW is an eager issuing approach. In beginning MPI_WIN_START or the first one-sided operation function if exists, the origin process is blocked until it receives all post messages from all processes in the target group. After that, all following MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs are issued immediately without queuing in the runtime. In ending MPI_WIN_COMPLETE, the origin process issues an additional 0-byte message to all processes in the target group to decrement the counter on their window to finish the current epoch. The eager approach always needs two synchronization messages.

Like the delayed issuing approach, our design for PSCW needs only one synchronization if the origin process owns operations issued to a target process, and requires two synchroniza-

tion messages if the origin has no operation issued to a target process. In MPI_WIN_START, processes in the origin group do nothing and return immediately. During the following MPI RMA epoch, each origin process performs as the delayed issuing at beginning: they queue up all posted operations in MPI runtime. When number of queued operations reaches the certain threshold, the origin process is blocked to wait for all the synchronization messages from target processes, and then issues all queued operations, and switches from delayed issuing mode to eager issuing mode. For the following MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs, the origin process issues them as early as possible. Like LOCK-UNLOCK, we avoid issuing another synchronization message by maintaining a pointer pointing to the last RMA operation for each target.

4.2.3 FENCE Synchronization

Like the above described LOCK-UNLOCK and PSCW synchronization, the current implementation of FENCE in MPICH also uses a delayed issuing approach. In the MPI_WIN_FENCE that initializes an MPI RMA epoch, all processes perform no synchronization at all and return immediately. All the following posted MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs are enqueued in MPI runtime. In the ending MPI_WIN_FENCE, each process first goes through all queued operations to decide that, for each other process rank i, how many operations have rank i as the target, and stores such information in an $O(P)$ array. After that, all processes perform a MPI_REDUCE_SCATTER communication with MPI_SUM operation on this array over the communicator of the window. After that, each process knows how many processes have operations targeting this process, and stores this information in the counter on that window. After that, each process issues out all queued operations, and the counter is decremented when all operations from the same origin process have been arrived, which is indicated by the packet header of the last operation from that origin process. Therefore, in the delayed issuing approach, only one synchronization, MPI_REDUCE_SCATTER, is required. Another choice for implementing fence is an eager issuing approach, in which all one-sided

operations are issued as early as possible. In the MPI_WIN_FENCE that open a new epoch, all processes perform a MPI_BARRIER synchronization over the communicator of the window. After that, every process issues out operations immediately. At the ending MPI_WIN_FENCE, all processes perform another MPI_BARRIER synchronization to ensure that no process leaves this MPI_WIN_FENCE before everyone have finished accessing the window. Therefore, in the eager issuing approach, two synchronizations (two MPI_BARRIER) are required. Our adaptive design for FENCE synchronization needs one synchronization (MPI_REDUCE_SCATTER) when number of operations is small, and two synchronizations (one MPI_BARRIER and one MPI_REDUCE_SCATTER) when number of operations reaches the certain threshold. As is shown in Figure 4.1, in the beginning MPI_WIN_FENCE, everyone does nothing and returns immediately. For all the following posted MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs, the process queues them in the runtime by default. If number of queued operations does not reach the threshold, which is the case for rank 0 in Figure 4.1, the process just goes into the ending MPI_WIN_FENCE and is blocked at the MPI_REDUCE_SCATTER synchronization in it. If the number reaches the certain threshold during the epoch, which is the case for rank 1 and rank 2 in Figure 4.1, the process is blocked at the MPI_REDUCE_SCATTER in the data movement operation function. Therefore, rank 1 and rank 2 are synchronized by MPI_REDUCE_SCATTER with rank 0. After the MPI_REDUCE_SCATTER completes, rank 0 issues all queued operations, whereas rank 1 and rank 2 also issue queued operations and the operations called after the MPI_REDUCE_SCATTER immediately without queuing any of them. When rank 1 and rank 2 enter the ending MPI_WIN_FENCE, they do not need to perform the MPI_REDUCE_SCATTER in it. At the end of MPI_WIN_FENCE, all processes need to be synchronized again by invoking a MPI_BARRIER. This strategy applies to ordered networks, or networks with remote completion mechanisms, in which processes can first wait for all remote completion events and then invoke MPI_BARRIER at end. On an unordered network without such mechanisms, MPI_BARRIER cannot ensure the completion ordering and all processes need to perform an all-to-all communication acknowledgment after complet-

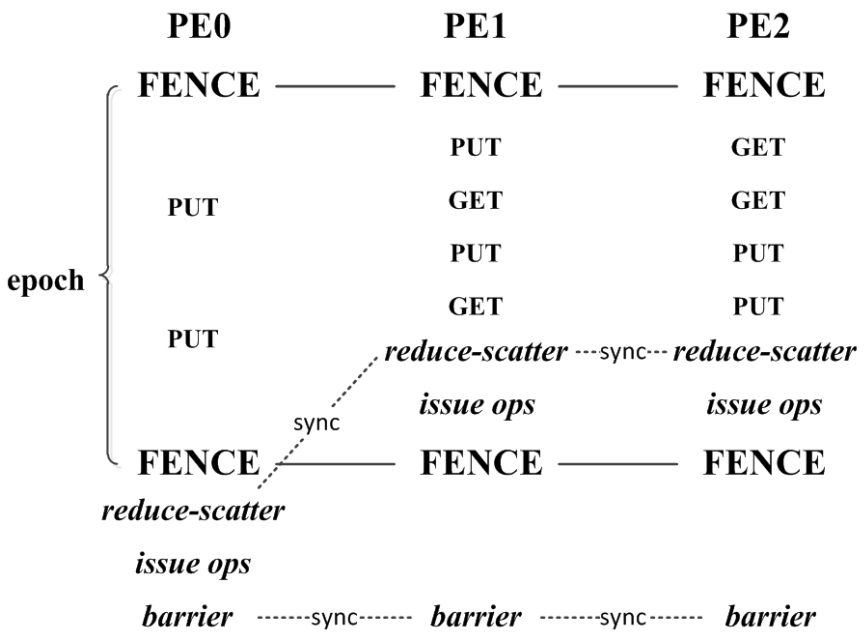


Figure 4.1: Adaptive FENCE

ing all operations. Note that full ordering for all data is not required to be imposed on the communication, just ordering of particular transfers with respect to others should be addressed.

If every process has small number of operations, which means FENCE is always in delayed issuing mode throughout the entire epoch, they only need one synchronization, which is the MPI_REDUCE_SCATTER in the second MPI_WIN_FENCE. An additional area in MPI_REDUCE_SCATTER is needed to indicate whether some process have called MPI_REDUCE_SCATTER before the closing MPI_WIN_FENCE; this is how the processes know whether a MPI_BARRIER synchronization is also required at end of fnMPI_WIN_FENCE.

4.2.4 Comparison with Existing Algorithms

For all three synchronization algorithms discussed above, the general approach for delayed issuing is to do nothing and return immediately in the beginning synchronization, enqueue all the following operations in MPI runtime, and to issues all operations and the ending synchronization call in the ending MPI RMA synchronization. The general approach for eager

issuing is to perform a blocking synchronization in the epoch opening MPI RMA synchronization call, to issue the following posted operations as early as possible, and to do another synchronization at end of MPI RMA synchronization. Compared with delayed issuing, eager issuing has more synchronization overhead and there is no optimization opportunity to aggregate or schedule the operations. However, eager avoids the cost of enqueuing operations and has the advantage of issuing operations immediately, in which they can arrive the target and be completed as early as possible. This is an especially desirable feature when there are large number of operations or the amount of data to transfer is large in MPI-based applications. Eager issuing also make it possible for overlapping communication and computation, which is impossible in delayed issuing. Our adaptive design combines features of both delayed issuing and eager issuing, while introducing a modest overhead in MPI runtime.

4.3 Experimental Evaluation

We implemented the adaptive approach based on the MPICH2 (1.4.1p1). For each synchronization mechanism, we also implemented the eager issuing to compare with delayed issuing and adaptive issuing approaches. Two metrics are used as the adaptive threshold: number of operations (h_{op}) and message size in each operation in bytes (h_{msg}). In the benchmarks, we set h_{op} as 10,000 and h_{msg} as 400. Our implementation implements the one-sided operations in AM-based way using a two-sided communication model to emulate the behavior of one-sided communication.

We use three kinds of microbenchmarks: single-operation test, many-operations test and overlapping test, which are described in Section 4.3.1 and Section 4.3.2 respectively. We measure the benchmarks on two different architectures: (a) an SMP machine with 4 Intel Core i5 CPU (2.67 GHz) and 8GB memory, we use it to emulate a architecture with a very fast interconnect hardware; (b) the "breadboard" cluster at Argonne National Laboratory on which each node has two Intel Xeon quad-core processors (2.66 GHz) and 16GB memory,

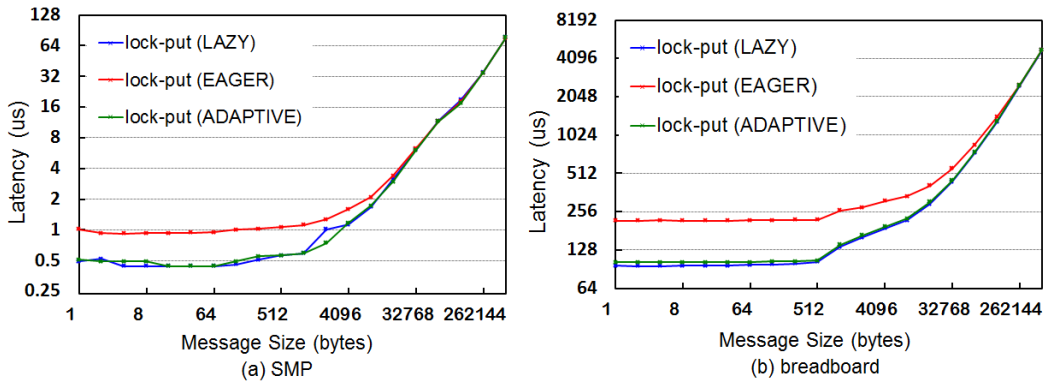


Figure 4.2: Single-op results on SMP and breadboard

and nodes are connected with Ethernet, we use it to examine the performance on a slow interconnect network hardware.

While all experiments in this section make use of a simple communication layer, the idea also apply to one-sided transports, particularly those transports that can implement the one-sided semantics as HW-based operations by directly exploiting RDMA features. It is notable that the delayed issuing mode allows the use of a single remote direct memory access operation, as long as the MPI semantics are observed, whereas the eager issuing mode allows the use of asynchronous communication for one-sided communication.

4.3.1 Latency Impact

Single Operation Results. We first measure the latency between two processes when single operation issued between synchronization calls. Figure 4.2 illustrates the MPI_PUT latency with LOCK-UNLOCK, with message size varying from 1 byte to 2^{18} bytes.

On SMP and breadboard, both adaptive approach and delayed approach perform better than eager approach when message size is small, due to the optimization for single short operation. Similar results are also observed for MPI_GET and MPI_ACCUMULATE operation.

Multiple Operations Results. We also evaluate the latency with increased number of short operations between synchronization calls. Figure 4.3 and Figure 4.4 presents the

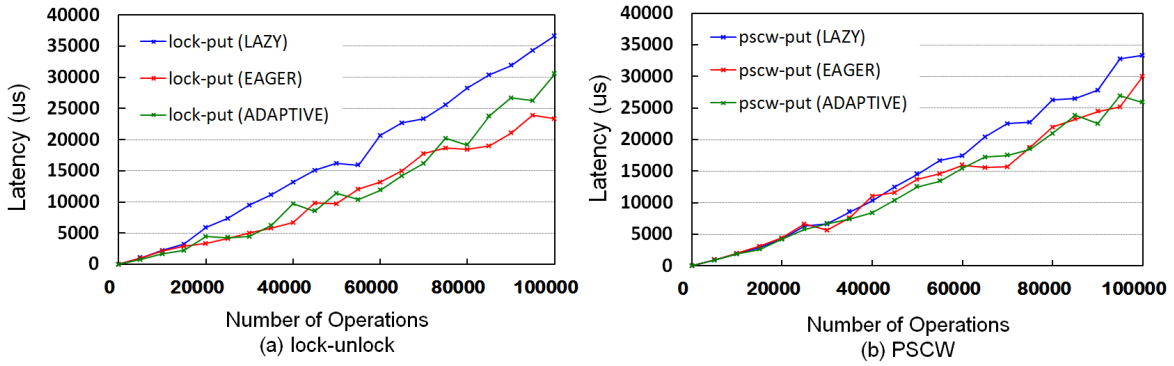


Figure 4.3: Many-ops results on SMP

MPI_PUT latency with LOCK-UNLOCK and PSCW on SMP and breadboard. Since FENCE is commonly used for with many neighbors, we did not measure its performance here.

On SMP, when number of operations is small, data transmission is very fast and there is no obvious difference between delayed issuing approach and eager issuing approach. When number of operations is relatively large, eager issuing approach and adaptive issuing approach perform better than lazy issuing approach. This is because of the extra queuing overhead in delayed issuing approach.

Figure 4.4 shows that on a slow network delayed issuing approach and adaptive issuing approach are better than eager issuing approach when number of operations is small, due to the extra synchronization cost in eager issuing approach; when number of operations becomes large, eager issuing approach and adaptive issuing approach perform better than delayed issuing approach, because the extra synchronization cost can be ignored due to the large amount of outgoing operations. Here we used number of operations as the threshold for adaptive issuing approach. Similar results are also observed for MPI_GET and MPI_ACCUMULATE operations.

4.3.2 Overlapping Impact

In this section we evaluate the overlapping performance by modifying the previous many-operations benchmark. We first evaluate the latency (t_1) for fixed number of operations plus

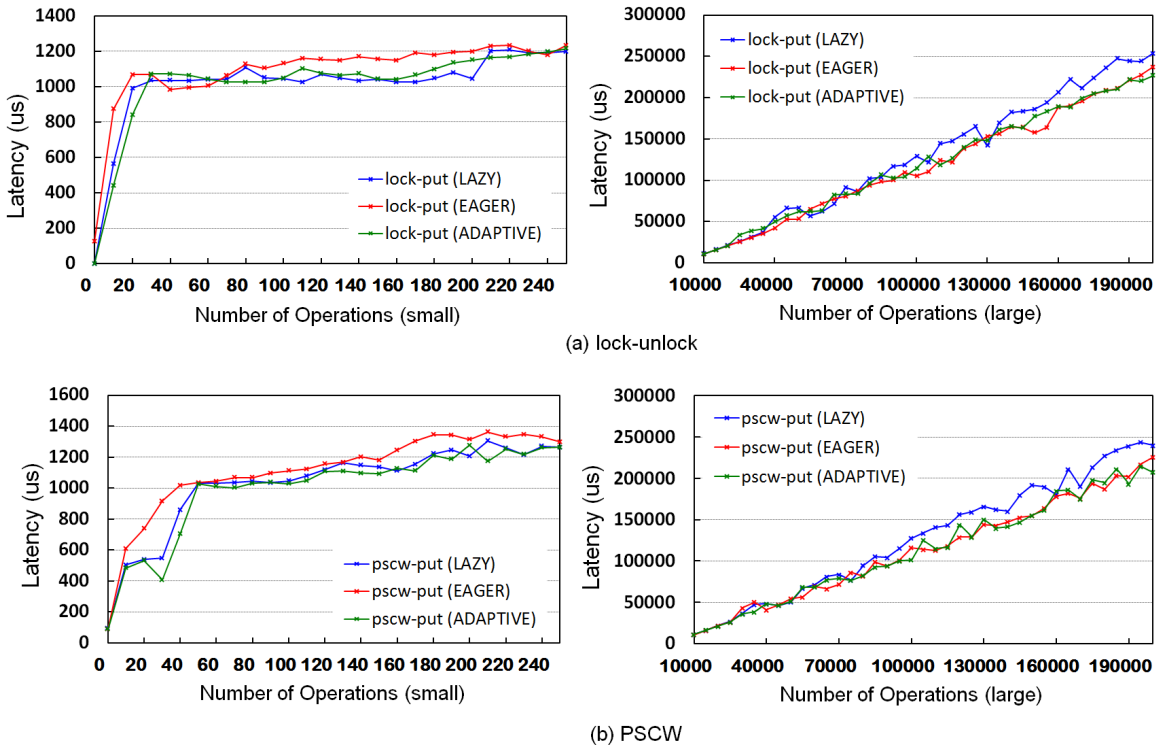


Figure 4.4: Many-ops results on breadboard

synchronization overhead between two processes, with no computation inserted. After that, we inserted some computation after one-sided operations with certain amount of computation corresponding to t_1. If the total execution time does not change, it means that all computation is overlapped by network transaction and the overlapping percentage achieves 100%; otherwise, we decreased the amount of computation until it can be completely absorbed by network communication.

Assuming time corresponding to the inserted computation is t_2, and the overlapping percentage equals to $\frac{t_2}{t_1}$. Table 4.1 illustrates the overlapping results of the adaptive issuing approach for MPI_PUT operation on breadboard, with number of operations being 4096.

Percentage for MPI_ACCUMULATE operation is similar with MPI_PUT operation, whereas percentage for MPI_GET operation is only half of it is for MPI_PUT operation. This is because for MPI_GET operation, the second synchronization needs to spend certain amount of time waiting for returning data, which is unable to be overlapped with local computation. For

Table 4.1: Overlapping results on breadboard

Message Size (bytes)	MPI_PUT (LOCK-UNLOCK)	MPI_PUT (FENCE)	MPI_PUT (PSCW)
2^{10}	30%	30%	15%
2^{11}	25%	25%	20%
2^{12}	60%	50%	50%
2^{13}	70%	60%	60%
2^{14}	70%	70%	65%
2^{15}	70%	75%	80%
2^{16}	70%	85%	80%
2^{17}	70%	70%	75%

delayed issuing approach, there is nearly no overlapping effects observed, and for eager issuing approach the overlapping percentage is similar to adaptive issuing approach. We did not run overlapping test on SMP, because the communication speed on SMP is very fast and computation is hard to be absorbed by one-sided communication.

4.3.3 Performance Impact in Mini-Apps

In this section, we measured the performance impact of adaptive strategy on Graph 500 benchmark [11] and MPPTEST benchmark [41]. Graph 500 benchmark is designed to demonstrate the suitability of systems for data-intensive applications by performing BFS on a randomly generated graph. The one-sided implementation of BFS in Graph 500 benchmark is implemented using MPI_WIN_FENCE and MPI_ACCUMULATE operations. Between each pair of MPI_WIN_FENCE calls, every process issues a lot of short operations to random other neighbors. MPPTEST benchmark includes an implementation of halo exchange, which can reflect common communication pattern in common simulation applications. In halo exchange, one process exchanges data with several other neighbors via multiple short data transmission between those processes. We use halo exchange to measure the impact of adaptive algorithm in PSCW. All benchmarks are run on breadboard.

Figure 4.5 illustrates the results of Graph 500 benchmarks and Figure 4.6 demonstrates

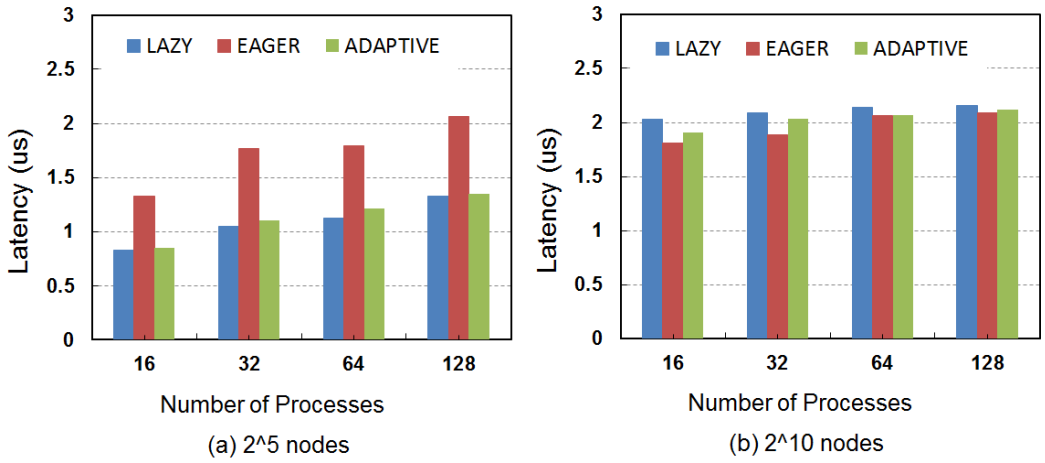


Figure 4.5: Graph 500 results on breadboard

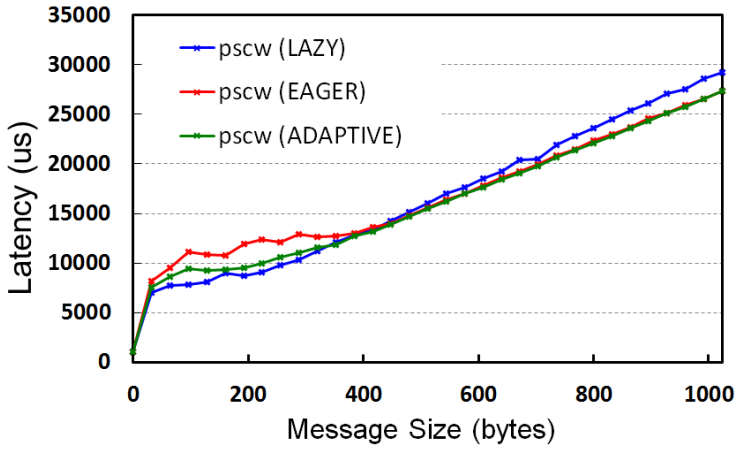


Figure 4.6: Halo exchange results on breadboard

the results of halo exchange with 128 processes and 8 neighbors. Both of them are run with three different issuing strategies. For halo exchange, we use message size instead of number of operations as the threshold in adaptive issuing approach.

4.4 Related Work

There are multiple studies regarding the implementation of one-sided communication in MPI. MPI implementations that support one-sided communication include MPICH [23], Open

MPI [24] and NEC [42]. Besides MPI, there are other programming models that provide one-sided communication paradigm, include CRAY SHMEM [43], ARMCI [44], GASNET [45] and BSP [46].

Some previous BSP works, especially paper [47] and [48], discuss the benefits of aggregating and scheduling communication operations to achieve better performance as well as avoiding message contention. Other papers, such as [49] and [50], described the design options and challenges in implementing one-sided communication in MPI. The authors in [51] have studied different optimization for reducing the synchronization overhead involved in implementing MPI one-sided communication. Designs for MPI RMA in InfiniBand clusters has been described in [52] and [53].In [54] and [55], the author describes a design for efficient passive synchronization using hardware support from InfiniBand atomic operations. In [56], authors discuss some performance guidelines for one-sided communication in MPI.

4.5 Conclusion

In this chapter, we describe the design and implementation of an adaptive issuing strategy used for one-sided communication in MPI. The current MPICH implementation uses a delayed issuing approach to issue operations, in which MPI runtime queues up posted operations and issues them at the epoch closing synchronization phase. This has certain advantages with respect to single short operations because of the reduced network operations. For large data transfers, however, issuing operations as early as possible is more beneficial than delaying them to the ending synchronization phase, and it can provide optimization opportunity for overlapping communication and computation.

Our design and implementation of adaptive issuing approach combine the advantages of both delayed issuing and eager issuing approaches with modest overhead. The performance results show that the adaptive approach performs as good as the delayed issuing approach for small data transfers and be able to achieve performance similar to eager issuing for

large data transfers. We also demonstrate the benefits of adaptive approach using Graph 500 benchmark and MPPTEST benchmark. This work is also included in author's Master thesis [57].

CHAPTER 5

Scalable Virtual Connection Initialization

One of the factors that can limit the scalability of MPI to exascale systems is the amount of memory consumed by the MPI implementation. In fact, some researchers believe that existing MPI implementations will consume a large fraction of the available system memory at exascale systems because there are several $O(P)$ data structure being used in MPI implementations. In this chapter, we describe our work focusing on reducing the memory consumption for virtual connection objects (VC objects) in MPI implementations.

5.1 Overview

Nowadays we have reached an era when the largest parallel systems in the world have achieved a few hundred thousand cores and will be approaching million-core systems. For instance, an IBM Blue Gene/Q system (Sequoia) at Lawrence Livermore National Laboratory in 2012 have more than 1.5 million cores. Community believes that on future systems we can expect systems with many millions of cores over the next 5 to 10 years. Another outstanding trend is that, even though the number of cores is increasing rapidly in recent years, the amount of memory provided on each core is not increasing respectively. As systems scale is increased to these sizes, many researchers wonder whether MPI runtime will be able to scale to such large-scale systems. An often-mentioned concern in the community is the potential memory consumption of MPI at large scale. It is widely believed that as the system size grows, the

memory consumed by MPI on each process will also grow rapidly. Considering the limited amount of memory per core and per node, it is believed that MPI will consume a large fraction of available memory and resources on Exascale systems. Effective solutions are necessary to prevent such situation coming.

To investigate these aspects and address potential problems, we designed and implemented specific optimizations for virtual connection objects inside MPI implementation to avoid linear memory growth. We describe these techniques and present experimental results demonstrating the memory savings achieved and the negligible impact on performance.

5.2 Linear Memory Growth in Virtual Connections

In most existing and popular MPI implementations, each process must maintain a state object for each other process with which it is actively communicating with. In MPICH, such state is kept in a virtual connection (VC) object associated with each of the remote processes. MPICH runtime creates one of these objects for each other process within MPI runtime, which means that, these VC objects consume $O(P)$ memory per process across an entire MPI application in the worst case. However, we note that the current design in MPICH is slightly more scalable than a naive design and implementation of maintain VC objects. In current MPI implementation, many buffers that are attached to the VC object are not allocated until communication actually happens with the target process corresponding to that VC. In other words, for connection-oriented communication substrates such as TCP network, such connections are not created until the communication really happens, therefore valuable saving operating system resources.

The VC implementation is also related another example where additional space is consumed in the trade-off with reduced access time. Each VC contains a area of "scratch pad" in which it is used by lower-level layer to store per-VC metadata. In order to clearly decouple lower-level layer from the upper layer stack of MPICH, space for such storage must

exist and allocated in MPI runtime. However, it would be sufficient for such space to be just large enough to maintain one single pointer, so that the lower-level layer could allocate and initialize a separate object and store a pointer in this minimal scratch pad region. Such approach would require an additional pointer dereference for the lower-level layer in order to access its own VC-specific metadata. Instead, by making the scratch pad region larger, this additional pointer dereference is saved for latency-sensitive data accesses that can be fit into the scratch pad region. Of course, tuning the size of this scratch pad is critical for performance of large P situation.

5.3 Lazy Initialization of Virtual Connections

To address the limitation of virtual connections, we developed an implementation that substantially re-designs the way VCs are managed. Under the new strategy, entire VC objects are created lazily only as needed instead of statically during MPI_INIT time. This change requires a fundamental shift in the way VCs are stored and accessed within MPI runtime.

In MPI runtime, MPI communicators are responsible for storing enough metadata so that they can determine which underlying process corresponds to a given rank in that communicator. This means that, given a communicator and a rank, the implementation must be able to generate a VC object. An naive implementation of this is to support this by a virtual connection reference table (VCRT). VCRTs consist of a dense array of VC references (VCR), each of them is indexed by communicator rank. The VCR is an opaque type, however, because of practical details of the interface, it must be implemented as a pointer to the underlying VC object. Each communicator stores a pointer to its VCRT and manipulates reference counts inside that VCRT table. In our design, the VCRTs on each communicator is replaced with a similar but more efficient concept: the Local Process ID mapping (or LPM). These objects perform a similar role with VCRTs, but rather than mapping communicator ranks to VCs directly and always via a dense array data structure, the LPM maps commu-

nicator ranks to LPIDs. This mapping decouples the upper-level layer, for instance MPI collective routines, from any notion of VCs that exist only at the lower-level layer. Unlike VCRTs, LPMs are truly transparent objects that are accessed only via function calls and macros. This design provides the chance to encode the communicator representation in a succinct, memory-efficient manner. Typical examples include using compression techniques that take advantage of domain-specific knowledge [34], more general compression methods [58]. Another example of domain-specific compression is supporting identity mappings, in which the LPID is equal to the communicator rank. Implementing this identity mapping is simple, given the new interface, and reduces memory usage on each process from $O(P)$ to $O(1)$ for communicators for which this mapping holds, such as MPI_COMM_WORLD. Conveniently, the LPM concept and interface also allow us to unify the representation of groups and the representation of communicator VC objects. Future improvements to this common LPM facility will provide dividends in both the group and communicator subsystems within MPICH. At the lower-level layer, VCs are obtained only via interfaces that refer to them by their LPIDs. This design allows true lazy instantiation and storage of VC objects, such as in a hash table, since upper-level layer no longer needs to hold pointers to all VCs.

5.4 Experimental Evaluation

In this section we provide substaintial experimental results to prove that a MPI implementation can restrict the use of memory within MPI runtime for scalable applications, without introducing a significant performance overhead. We first look at several microbenchmarks, after that we evaluate several application-based benchmarks. All results were gathered on the "Fusion" cluster at Argonne National Laboratory. Each node consists of two Intel Xeon X5550 quad-core processors, and the nodes are connected by Mellanox QDR Infiniband.

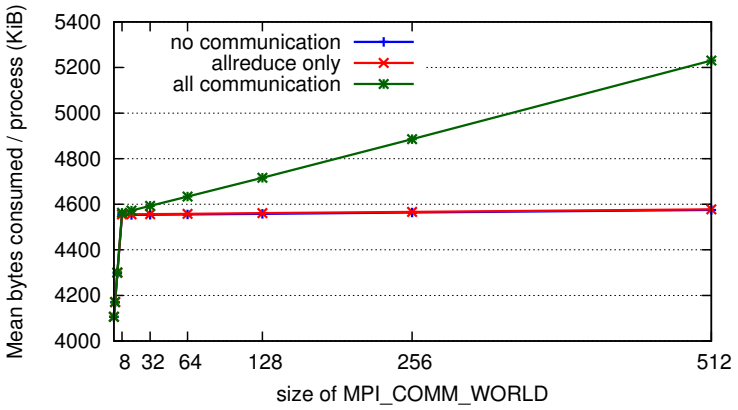


Figure 5.1: Per process memory consumption

5.4.1 Scalable Memory Use

In order to varify the memory usage of the strategy proposed in the previous section, we use three microbenchmarks here which extract the basic communication behavior from sophisticated application scenarios. These microbenchmarks respectively perform (a) no communication, (2) scalable communication (a single MPI_ALLREDUCE), and (c) nonscalable communication (pairwise communication between all processes). Furthermore, the MPI library was configured to allow for memory consumption measurements to be taken. Figure 5.1 illustrates the results of running these experiments with the "lazy" initialization approach enabled. As we expected, the "no communication" and "allreduce" benchmarks consumed no additional memory on each process as the job size grows, whereas the "all communication" benchmark shows that memory usage per process is increased linearly with job size. This increase indicates an $O(P^2)$ memory consumption in the entire system.

5.4.2 Performance Impact

Figure 5.2 shows the evaluation of bandwidth and latency with the stable "Trunk" version of MPICH as a reliable baseline, and the strategy configured to use an eagerly constructed dense array ("Eager") or lazily constructed sparse hash table ("Lazy") for storage of VC objects. A slight decrease at large-message bandwidth and a slight increase at small-message latency

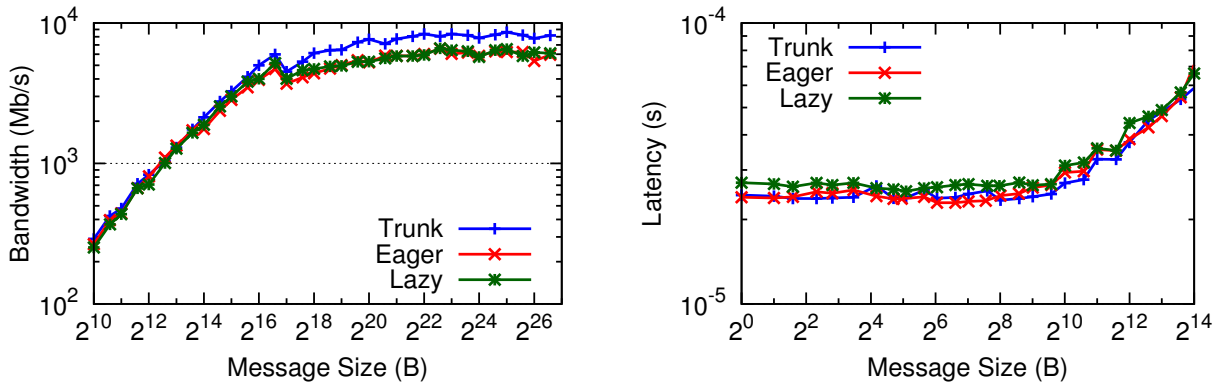


Figure 5.2: Netpipe ping-pong performance results

can be observed from the experiment results. We note that the prototype code has not been tuned to achieve the best performance; we expect to eliminate most of this performance gap in future effort.

5.4.3 Application Impact

In this section, we measured the impact of our changes on scalable applications by evaluating the performance and memory consumption behavior of certain NAS Parallel Benchmarks [59] and the Sequoia AMG benchmark. Both of them are representative of application behavior. These codes are also well known and commonly used to represent the behavior of many real-world scientific applications. All of these benchmarks exhibit scalable communication patterns; which means that the number of communication peers keeps flat or increases slowly when job size increases. Table 5.1 lists the performance impact and memory usage per process of our techniques when applied to the CG and MG class D NAS Parallel Benchmark. The benchmarks were run with the same three configurations from Figure 5.2. MPI memory consumption is reduced in the lazy approach by about 550 bytes per process (≈11%), at a cost of less than 5% in performance. Figure 5.3 illustrates memory consumption on each process versus job size when running the Sequoia AMG benchmark [60] with both eager and lazy VC initialization strategies. The benchmark was to solve a Laplace-type problem4 with two different three-dimensional processor layouts. The first layout was cubic, for example

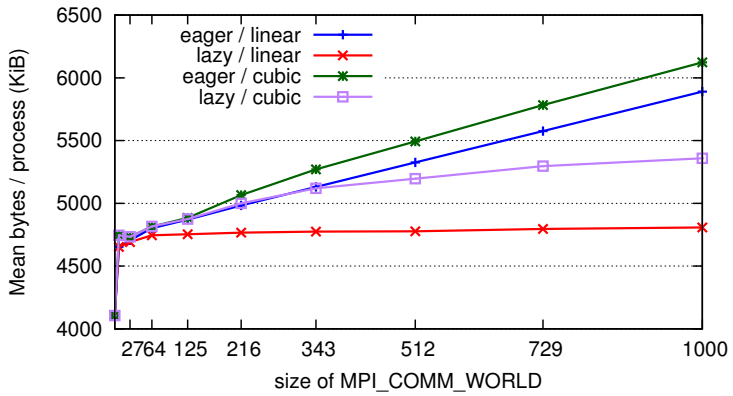


Figure 5.3: Per process memory consumption in Sequoia AMG benchmark

36 processes organized as $Px \times Py \times Pz = 6 \times 6 \times 6$. The plot demonstrates a substantially growing of memory consumption for this case when lazy VC initialization is used. The second layout was entirely linear, for example by $36 \times 1 \times 1$. This layout has far fewer communication peers, which results in the expected almost flat memory usage on each process.

Table 5.1: Performance of selected NAS Parallel Benchmarks

Benchmark	**MPI**	**Time (s)**	**Memory / Process (kiB)**
cg.D.512	Trunk	536.77	5,149.2
	Eager	520.55 (-4.02%)	5,144.7 (-0.09%)
	Lazy	556.82 (+3.74%)	4,588.2 (-10.89%)
mg.D.512	Trunk	18.69	5,154.2
	Eager	19.19 (+2.68%)	5,154.3 (+0.00%)
	Lazy	19.49 (+4.28%)	4,602.3 (-10.71%)

5.5 Conclusion

In this chapter, we propose a new implementation of MPI virtual connection objects, and show that an MPI implementation can be constructed so that memory use grows slowly as the number of processes increase and that the performance cost for a real application is low. This work is a collaboration between the author of this thesis and David Goodell [61] and the implementation was completed by both of them. Specifically, David Goodell completed

the design of the work and the implementation of upper-level framework, whereas the author completed the implementation of lower-level framework, including lazily initializing VC objects and storing VC objects in a hash table.

CHAPTER 6

Generalized MPI-Interoperable Active Messages

In recent years, irregular and data-intensive applications have become increasingly important in many areas, however, traditional data movement approaches like two-sided communication for scientific computation are not well suited for such kind of new applications. The Active Messages (AM) paradigm is an alternative communication paradigm that is better suited for such applications by allowing computation to be dynamically moved closer to the data. Considering the wide usage of MPI in scientific computing and many other applications, enabling an MPI-interoperable AM paradigm would allow traditional applications to incrementally start utilizing AMs in portions of their applications, thus avoiding the programming effort of rewriting the entire MPI-based irregular application. In this chapter, we present a design and implementation of a generalized framework of MPI-interoperable AM, called MPI-AM, which can provide a general semantics of Active Messages integrated with MPI to accommodate a wide variety of computational patterns in irregular applications. Together with a set of new APIs, we describe a detailed description of the correctness semantics of MPI-interoperable AM, optimization techniques and asynchronous processing, and a reference implementation that demonstrates how various implementation choices affect the flexibility provided to the MPI implementation and consequently its performance.

In the generalized framework for MPI-interoperable Active Messages, we first propose a

new set of functionality and semantics that do not extend the existing MPI_ACCUMULATE-like operations but still maintain complete compatibility with the MPI-3 standard. The new semantics allow the implementation to achieve better performance for example by controlling streaming and data usage granularity, concurrency capabilities among AMs, and ordering semantics.

6.1 Overview

Many new, irregular and data-intensive applications have become popular in recent years in various domains such as bioinformatics, computational chemistry and social network analysis. A fundamental and common characteristic of these applications is that they involve a large amount of transferred data, possibly in irregular patterns, requiring computation and data movement to be carefully balanced in order to achieve high performance. Traditional programming models, such as MPI two-sided communication or collective communication, that were designed for environments where computation is regular and communication cost is significantly larger than the data movement cost, are not well suited for such applications. Alternative programming frameworks are needed.

The Active Messages (AM) model [13] is a parallel programming paradigm that is more suitable for irregular applications. It allows a process to specify a function handler to be executed when an active message arrives the target side, therefore relieving the responsibility of the target to explicitly receive and process the incoming message. Such a model can be more natural to use in some scenarios in irregular applications. A combination of the traditional MPI_SEND / MPI_RECV or MPI_PUT / MPI_GET models to move data and an AM-based model to move computation can provide applications and high-level libraries capabilities to efficiently and dynamically balance their computation and data movement in order to achieve high performance.

In this chapter, we present a generalized framework for MPI-interoperable AMs. Specif-

ically, we propose a new set of functionality and semantics that does not rely on existing MPI_ACCUMULATE-like operations but still maintain complete compatibility with the latest MPI-3 standard. The new semantics allow the implementation to achieve better performance, for instance by controlling message streaming and data usage granularity, concurrency capabilities among AMs, ordering semantics and asynchronous processing. In addition to the design description, in this chapter, we also present a reference implementation of the generalized framework and experimental evaluation results demonstrating the performance impact of the various semantic choices, different buffer supplement strategies and message streaming configuration, and asynchronous processing.

6.2 Background and Related Work

The concept of Active Messages (AM) was proposed by von Eicken et al. for Split-C in [13] in 1992 and has been used internally to implement various communication libraries and runtime systems, such as MPI implementations, Co-Array Fortran (CAF) and Unified Parallel C (UPC). With the capability of AM, the sender of a message can specify a message handler to be executed at the receiver side upon arrival of that message. When the message arrives, the corresponding AM handler is triggered to perform certain user-defined computation with the data in that message and the data on the target side. Unlike traditional MPI two-sided communication, the application on the receiver side does not need to explicitly make a function call in order to receive that message. Previous libraries that support such capability of AM include GASNet [62], IBM DCMF [63], IBM LAPI [64] and IBM PAMI [65]. While they are popular in high performance computing, those libraries either do not maintain runtime compatibility with MPI infrastructures, or are too low-level and platform-specific. Existing MPI-based applications cannot directly use them without duplicating runtime resources such as internal buffers or asynchronous threads. Since existing AM libraries and interfaces are too low-level for application to directly use, there is previous work that has been done on

supporting AM on top of the MPI library, so that existing MPI applications can gradually utilize the AM communication paradigm only when necessary. Previous works of MPI and AM include AM++ [66] and AMMPI [67]. While they are portable implementations, those libraries are too restricted in a number of ways, including inability to marshall and demarshall datatypes, lack of asynchronous progress and absence of explicit semantics for memory consistency, ordering, concurrency of AMs, and atomicity.

Like what we introduced in Chapter 2, MPI programming model and runtime systems are widely portable and support two-sided, collective and one-sided communication. In MPI one-sided communication, the originator of the operation (e.g. MPI_PUT, MPI_GET and MPI_ACCUMULATE) specifies the memory location of the buffer at the remote target where the data is to be sent to or received from. This "one-sided" style of communication is similar with AM in which the originator of AM specifies the location of the buffer at the remote target where the AM is going to interact with. On the other hand, in MPI_ACCUMULATE-like operations, the originator specifies a certain type of computation (e.g. MPI_SUM, MPI_MIN, MPI_MAX) that is going to be performed on the remote side. This is quite close to the AM handler in Active Messages. However, MPI_ACCUMULATE-like operations only supports a limited predefined set of computation and does not support user-defined computation. It is natural to consider extending MPI_ACCUMULATE-like operations to support user-defined computation in order to achieve AM capability in MPI, however, there are several inevitable restrictions with it. In this thesis, we call such design as "accumulate-style AM" and we will discuss those restrictions in Section 6.3.

6.3 Restrictions of Accumulate-Style Active Messages

The semantics of MPI_ACCUMULATE-like operations, including MPI_ACCUMULATE, MPI_GET_ACCUMULATE and MPI_FETCH_AND_OP, are very close to the concept of AM. MPI_ACCUMULATE-like operations allow users to specify certain calculations to be performed

on remote memory. However, they only support a limited set of predefined computation and do not support user-defined computation. To allow for more flexible computation on remote processes, one solution is to extend MPI_ACCUMULATE-like operations to support user-defined function in MPI, i.e., to implement as accumulate-style AMs.

In MPI-3 standard, user-defined function is proposed to be used with MPI_REDUCE-like collective communication. We can use the same mechanism to allow user to define user-defined function for MPI_ACCUMULATE-like operations. The ISO C prototype of user-defined function is as the following: `typedef void MPI_User_function (void* invec, void* inoutvec, int* len, MPI_Datatype* datatype)`. Arguments `invec` and `inoutvec` represent two buffers that the function combines, whereas arguments `len` and `datatype` describe the data amount and data layout of those buffers. Each invocation of the function leads to the pairwise execution of user-defined computation on elements in buffers. The results of the function are returned to the buffer specified by `inoutvec`. After defining the user function, user needs to call MPI_OP_CREATE routine to bind the function to an MPI operation handle that can be subsequently passed to MPI_ACCUMULATE-like operations.

Even though enabling MPI_ACCUMULATE-like operations to support user-defined functions would achieve a functionality of "one-sided" style communication and allow user to specify computation on the remote side, MPI_ACCUMULATE-like operations were not originally intended for AMs, and their semantics do not quite match with what AMs need. Consequently, accumulate-style AM, which are based on MPI_ACCUMULATE-like operations, has several restrictions when being used as an AM framework in practice. We will discuss those restrictions in the following sections.

6.3.1 Data Access

One restriction in using accumulate-style AMs comes from the way it represent the data layouts. More specifically, the user-defined function which was originally intended for MPI_REDUCE-like operations in MPI accepts a single data layout, i.e. a single datatype and

count. Therefore, both the input and output buffers must have exactly the same layout when using with that user-defined function. While this requirement is suitable for MPI_REDUCE, where multiple input arrays are reduced into a single target array, it is too restrictive for the usage of AMs. For instance, considering an application where the data on the target process is an array of bins of containers representing value ranges. When the target process receives an AM with an integer that falls into one of those bins, the corresponding counter is incremented. In such a scenario, the AM input data contains a single integer, while the AM handler needs access to the entire array of bins to do the necessary computation. Such kind of computation cannot be handled by accumulate-style AMs.

A similar restriction arises with respect to the data that is returned back to the origin process. While both MPI_GET_ACCUMULATE and MPI_FETCH_AND_OP provide such capability, their semantics require it to return the original data at the target before the modification is performed. Therefore, the AM cannot flexibly return any arbitrary user-specified data. One example where such capability is necessary is DNA sequence assembly applications (e.g., SWAP [1] and Kiki [2]). These applications rely on storing databases with large data volume of DNA sequences on distributed-memory system. A process that needs to search for a DNA sequence in the database would issue the query sequence to the target as an AM, which would then search through its database and return the matches. Although the AM requires access to the entire data on the target process, it does not necessarily require all of this data to be returned to the origin. Only the matches need to be returned. To emulate such a function, SWAP currently uses MPI_SEND / MPI_RECV with threads, potentially wasting cores waiting for incoming requests. Kiki uses dedicated "server processes" that only process such messages but perform no application computation, again wasting some user-defined number of cores for these processes. Applications in other domains, such as MADNESS [4] (computational chemistry), are similarly restricted and emulate AM functionality using MPI_SEND / MPI_RECV with threads.

6.3.2 Message Segmentation and Temporary Buffers

The MPI-3 standard has not required the MPI implementation to have any temporary buffers. That is, the application cannot assume that there are any temporary buffers within MPI implementation for communication and has to perform correctly when no such buffers are available in MPI runtime. For example, although most MPI implementations use "eager buffers", also referred to internal temporary buffers, when communicating small messages and application (or user) buffers through a "rendezvous" synchronized hand-shake when communicating large messages, the MPI-3 standard does not expose such modes to the application. Thus MPI implementations is free to experiment with different buffering techniques in order to improve performance.

This concept is also apparent in MPI_ACCUMULATE-like operations that guarantee atomicity only at the granularity of predefined datatypes. Therefore, an MPI implementation that has no internal buffers can segment operations into multiple smaller operations, which is potentially at the granularity of one predefined datatype per operation, and issue them one by one. Since MPI_ACCUMULATE-like operations permits only predefined computations, the MPI implementation knows these operations beforehand and can implement them appropriately in order to compute on segmented data. For accumulate-style AM, however, because the computation is now defined by the user's application, the MPI implementation is unable to know what an appropriate segmentation granularity would be. Segmenting data to the granularity of a predefined datatype might be too restrictive for the irregular application. For instance, in the DNA assembly example given in Section 6.3.1, if the MPI runtime devides the input DNA string sequence to the granularity of individual characters, the AM user function cannot search for the entire string in the target database with this input information. On the other hand, if the MPI runtime had to send the entire input data to the target process, it will be required to buffer arbitrarily large input data internally, which causing the memory usage problem.

6.3.3 Lack of Concurrency

In accumulate-style AM, semantic of concurrent execution of AMs is not well defined. In an environment where multiple origin processes simultaneously trigger AMs on the same target process, or where the same origin process triggers multiple concurrent AMs on the same target process, can the MPI implementation simultaneously execute those different AMs? MPI_ACCUMULATE-like operations are atomic at the granularity of predefined datatypes. That is, if two such operations target at the same memory location on the same target, the MPI implementation will ensure that these updates do not interfere with each other. With user-defined operations, however, MPI can no longer keep track of such atomicity on target window. A conservative implementation of accumulate-style AM would be to serialize all AMs computing on overlapping memory locations and process them one by one, thus forcing no concurrency in execution. However, such an implementation would bring expensive performance overhead. For instance, for AMs that only need to read the target data but do not need to update it, no such atomicity is required, and the lack of concurrency can hurt performance a lot.

6.3.4 Interoperation with Other MPI Messages

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

6.4 Design and Implementation of Generalized MPI-Interoperable Active Messages Framework

In this section, we present a design and implementation for generalized MPI-interoperable AMs that addresses the shortcomings discussed in Section 6.3. The design leverages the MPI RMA interface but is no longer based on MPI_ACCUMULATE-like operations but proposes a much more generalized framework of AM capability.

6.4.1 Data Streaming in Active Messages

The MPI implementation can invoke a user-defined AM function multiple times to handle a single large message. If the origin and target buffers are arrays, then the user-defined AM function may be called to process smaller portions of the array. This allows the message reception to be pipelined with the execution of the user-defined AM function, meanwhile reduces the memory required to stage the incoming message. We implemented pipelining by first receiving a chunk of the incoming data then calling the user-defined function with the data received from origin and the data in the target buffers. Each chunk is defined as one "segment".

Each AM contains multiple segments as represented by the argument num_segments in AM handler, as illustrated in Figure 6.2 (AM handler is described in Section 6.4.3). The MPI implementation is allowed to split an AM at any system-dependent size at the granularity of one segment, which means that such system-dependent size must be a multiple of segment size. Such capability is helpful, for instance, when the user or the MPI implementation does not have enough temporary buffers to stage the entire input and output data of one AM handler. Even when enough temporary buffer space is available, the MPI implementation can still choose to pipeline the data transfer with the AM computation in order to improve the performance. One significant difference with MPI_ACCUMULATE-like operations is the gran-

ularity of segmentation. As described in Section 6.3, in MPI_ACCUMULATE-like operations, the MPI implementation is allowed to segment a message at the granularity of predefined datatypes. With MPIX_AM, the runtime system cannot know the minimum granularity of segmentation for each AM, therefore it must be specified by the user via num_segments in AM handler. For example, in the example of DNA sequence assembly discussed in Section 6.3.1, the minimum granularity for input data to AM handler is one DNA sequence, therefore, the user should define one segment as one DNA sequence and specify num_segments at least as one in order to trigger the execution of AM handler on target side.

6.4.2 Data Buffering Requirements

In MPI-AM framework, each AM handler is associated with a temporary input buffer and a temporary output buffer that are valid only within the execution of AM handler. An important question here is who is responsible for allocating and managing such temporary buffers. Most previous AM frameworks assume that the runtime system (in this case, MPI) would allocate and maintain such internal buffers. However, since the runtime does not know the minimum granularity maximum consumption of AM handler, and there is no upper bound on how much memory an AM would need in order to perform computation, that is not a reasonable assumption and should be carefully defined.

To this end, we propose two new routines: MPIX_AM_WIN_BUFFER_ATTACH and MPIX_AM_WIN_BUFFER_DETACH, as illustrated in Figure 6.1. MPIX_AM_WIN_BUFFER_ATTACH allows the user to provide certain amount of temporary buffer in user's memory space to the MPI implementation in order to accommodate data in incoming AMs. MPIX_AM_WIN_BUFFER_DETACH reclaims the buffer from the MPI implementation. While the MPI implementation might also internally provide additional temporary buffers, the MPI application should not assume the availability of such internal buffers but should always explicitly provide enough user buffers beforehand.

The size of the user-provided buffer must be large enough to serve input and output

```
MPIX_AM_WIN_BUFFER_ATTACH (buffer, size, win)
  IN    buffer      initial buffer address (choice)
  IN    size        buffer size, in bytes (integer)
  IN    win         window object (handle)

MPIX_AM_WIN_BUFFER_DETACH (buffer, win)
  IN    buffer      initial buffer address (choice)
  IN    win         window object (handle)
```

Figure 6.1: Prototype of AM buffer attach / detach routines

buffers corresponding to at least one AM segment. On the other hand, the temporary user buffer is shared by AMs from all origin processes. Therefore, in the implementation of MPI-AM framework, the origin process needs to perform appropriate synchronization beforehand with the target process in order to "reserve" portion of the user buffer before it can issue the AM data. Furthermore, the user-provided temporary buffer must be large enough to accommodate the target input and output data in the user-described, potentially sparse, data layout, which means that the temporary buffer should be at least the "MPI true extent" of the target input and output datatype counts. The true extent can be returned using MPI_TYPE_GET_TRUE_EXTENT. While MPI-Am allows sparse datatypes, which have a small size but a large extent, to be used in the AM handler, such datatypes are discouraged in practice because of the large unnecessary memory usage they consumes.

Apart from user buffers, in the implementation of MPI-AM, we also provide internal system buffers for the usage of AMs. The internal buffers are not shared among origin processes but are allocated separately for each origin process, therefore the origin side does not need to synchronize with the target in order to reserve buffers beforehand. When internal buffers are used up, the MPI runtime will switch to use user-provided buffers.

6.4.3 Generalized Interface

In this section, we introduce the generalized interface for MPI-AM framework, including user-defined AM handler, AM trigger routine, and handler creation routine and handler

registration routine. We also include an example showing how to use MPI-AM in the program.

User-Defined Active Messages Handlers. We first propose a new prototype of the user-defined function handler, MPIX_AM_USER_FUNCTION, that would execute when an AM arrives at a target. In this thesis we refer to that handler as the "AM handler". The prototype of it is illustrated in Figure 6.2.

In the high-level working model, the user defines a AM handler with the MPIX_AM_USER_FUNCTION prototype and creates an MPI operation handle with that handler using MPIX_AM_OP_CREATE. Once an MPI operation handle is created, the user collectively registers the operation handles across a group of processes where every process provides a functionally equivalent AM handler.

The handler executes in user context, as opposed to an interrupt or signal context, and has access to three buffers: input buffer (specified by arguments with prefix input_), persistent buffer (specified by arguments with prefix persistent_, and output buffer (specified by arguments with prefix output_). Data in the input buffer is provided by the origin; data in the output buffer is generated during the execution of AM handler and is returned to the origin at the completion of the AM. Both the input and output buffers are private to the AM handler and are temporary. That is, neither the buffer nor its content is valid outside of the AM handler. The MPI implementation can stage such data in temporary buffers and discard the buffers or the data in those buffers at the end of the AM handler. The persistent buffer points to the part of the target window that the AM handler has access to and is persistent across AMs. That is, the buffer is available and valid outside of the AM handler as well. The AM handler can update the data in the persistent buffer. Argument num_segments specifies the granularity of data segmentation and argument segment_offset specifies the starting offset in segment units for current AM. Data streaming and segmentation is described in Section 6.4.1.

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

Figure 6.2: Prototype of AM handler

Registration of Active Messages Handler. We propose four new routines for AM handler creation and registration, as illustrated in Figure 6.3. MPIX_AM_OP_CREATE is similar with MPI_OP_CREATE in MPI-3 standard [8] which associates an user-defined function to an operation handle. Difference is that MPIX_AM_OP_CREATE associates user-defined AM handler with an MPI operation handle whereas MPI_OP_CREATE associates user-defined function (used in MPI_REDUCE-like operations) with an MPI operation handle. MPIX_AM_OP_FREE is used to free the MPI operation handle.

MPIX_AM_OP_REGISTER is a call among all processes that are in the same window. When the routine is called, the operation handle and a AM handler index are passed in and run-

```
MPIX_AM_OP_CREATE (user_fn, op)
  IN      user_fn          AM user defined handler (function)
  OUT     op               operation (handle)

MPIX_AM_OP_FREE (op)
  INOUT   op               operation (handle)

MPIX_AM_OP_REGISTER (op, index, win)
  IN      op               operation (handle)
  IN      index            index for this handler (integer)
  IN      win              window (handle)

MPIX_AM_OP_DEREGISTER (op, win)
  IN      op               operation (handle)
  IN      win              window (handle)
```

Figure 6.3: Prototype of AM creation and registration routines

time stores them in a hash table on that window. This registration routine is collective to ensure that the operation is registered on all processes including potential targets before any origins starting issuing active messages. Because operation handles are local objects, an AM handler index is necessary for AMs to identify handler functions on remote processes. In this way, the user-defined operation can be "available everywhere". It is user's responsibility to guarantee that operations defined on different processes with the same AM handler index have equivalent functionalities. MPIX_AM_OP_DEREGISTER is used to de-register the AM handler from the window.

Two handlers are "functionally equivalent", if one can be executed instead of the other to get an equivalent result. For architectures that use different byte-widths or byte-ordering for datatypes, the MPI implementation will need to do necessary data transformation before executing the AM handler. Thus, the MPI operation represents a group of functionally equivalent handlers distributed across processes. The functional equivalence of the handlers makes it valid for the MPI implementation to replace one handler with another. For example, instead of transmitting the AM to the target process, the MPI implementation can fetch the target data locally and execute the AM using the local equivalent handler.

Such an approach is particularly useful for shared-memory systems where the "remote data" might be directly visible to the process through shared memory.

Active Messages Trigger. We propose a new routine for issuing AMs, called MPIX_AM, that manages the data and computation associated with an AM, as illustrated in Figure 6.4. This section describes the data associated with the AM. The computational function handler is represented by op.

MPIX_AM allows the user to specify arguments associated with totally five buffers on origin side and target side: origin input buffer (specified by arguments with prefix origin_input_), target input buffer (specified by arguments with prefix target_input_), target persistent buffer (specified by arguments with prefix target_persistent_), target output buffer (specified by arguments with prefix target_output_), and origin output buffer (specified by arguments with prefix origin_output_).

Argument origin_input_addr provides the buffer on the origin side associated with the input data, while arguments origin_input_segment_count and origin_input_segment_datatype represent the data layout. The target input buffer stages data that would be transmitted to the AM handler as input data. When the data is transferred to the target process, we allow the data representation to be modified to a different layout as represented by target_input_segment_datatype. This capability is useful for applications that use sparse data layouts on the origin for the input buffer (e.g., elements on the nonleading dimension of a matrix), but can represent them in a more space-concise format at the target (such as a contiguous list of elements). Note that the type signature of origin_input_segment_datatype does not need to match that of target_input_segment_datatype. However, there must exist a non-negative integer "N", where the type signature of origin_input_segment_count $\times$ origin_input_segment_datatype should match that of "N $\times$ target_input_segment_datatype". In other words, the runtime system should be able to represent the data in each segment using a collection of target_input_segment_datatype elements. "N" is internally calculated by the MPI implementation.

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

Figure 6.4: Prototype of AM trigger routine

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

The target persistent buffer represents data that already exists at the target process and is used within the AM handler. Arguments target_persistent_disp, target_persistent_datatype and target_persistent_count represent the portion of the data that is accessed by the AM. While the AM can represent the entire target memory window using these parameters, an accurate representation of the required target data can allow the MPI implementation to optimize data movement in some cases (e.g., where the target data is smaller than the origin data, by fetching the target data and computing on it locally) or better identify opportunities for concurrency or out-of-order execution of AMs. Argument num_segments specifies the granularity of data segmentation for the current AM. The data streaming and segmentation is described in Section 6.4.1.

MPIX_AM specifies the computation to be performed on the data but does not specify where the computation actually happens. In the other words, the AM origin process and target process only describe the locality of the data. The MPI implementation can choose to execute the AM computation on the target side, on the origin side, or at any other location. While forcing the computation to occur at the target side would allow applications to be able to better control the computational resource, it would take away the MPI implementation's capability to do trade-off between computational locality and data movement, for instance by moving the target data to the origin process and computing on it locally. Unfortunately, there is no clear winner between these two choices. In our design of

MPI-AM, we allowed the MPI implementation to have more flexibility at the cost of fewer guarantees on the computational locality.

Code Example. In this section, we describe an example code using the interface of MPI-AM described above. This example can take advantage of generalized MPI-interoperable AM.

In the code illustrated in Figure 6.5, my_sum represents the user-defined AM handler that the application would like triggered when an AM arrives. The function itself sums up the data at the origin side. The application first creates an operation sum_op out of the user-defined function handler using MPIX_AM_OP_CREATE. After he operation is created, the application collectively registers it across all processes on the window using MPIX_AM_WIN_OP_REGISTER. Once the operation is registered, rank 0 issues a single AM to rank 1 within a LOCK-UNLOCK epoch. We note that the application attached a user buffer of 100 bytes to rank 1 beforehand, in preparation for the AM.

6.4.4 Workflow of MPI-Interoperable Active Messages

The workflow of the MPI-AM framework is illustrated in Figure 6.6. With the routines proposed in this section, 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. We note that target input buffer and target output buffer are internal buffers associated with each AM handler whereas the rest buffers are public buffers. When MPIX_AM is called, the origin input data is sent to the target process and is staged in the target input buffer. This staged data serves as the input data to the AM handler, and the handler stores its output data into the target output buffer. Once the computation in the handler is finished, the output data is returned back to the origin output buffer. The target persistent buffer stores the data that already exists at the target's window and is accessed within the AM handler. All modifications on this buffer can be seen by future AM

```
1  void my_sum(void *a, MPI_Aint *a_seg_off, int *a_seg_cnt,
2              MPI_Datatype *a_dtp, void *b, int *b_cnt,
3              MPI_Datatype *b_dtp, void *rst, int *rst_seg_cnt,
4              MPI_Datatype *rst_dtp)
5  {
6      int i;
7      int *in_a = (int *)a;
8      double *in_b = (double *)(b + (*a_seg_off) * 2);
9      double *out_rst = (double *)rst;
10
11     for (i = 0; i < (*a_seg_cnt); i++)
12     {
13         /* Each output element depends on many target elements */
14          *out_rst = (double) (*in_a + *in_b + *(in_b+1));
15          in_a++;
16          in_b += 2;
17          out_rst++;
18     }
19 }
20
21 int main(int argc, char* argv[])
22 {
23     ...
24
25     /* attach user buffer to window */
26     if (rank == 1)
27         MPIX_Am_win_buffer_attach(user_buf, 100, win);
28
29     /* create and register user operation */
30     MPIX_Am_op_create(my_sum, &sum_op);
31     MPIX_Am_win_op_register(sum_op, 1, win);
32
33     /* attach buffer and issue active message */
34     if (rank == 0) {
35         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
36         MPIX_Am(a, 10, MPI_INT, rst, 20, MPI_DOUBLE, 100, 1,
37                 MPI_INT, 0, 2000, MPI_DOUBLE, MPI_DOUBLE, sum_op, win);
38         MPI_Win_unlock(1, win);
39     }
40
41     /* detach and free user operation */
42     MPIX_Am_win_op_deregister(sum_op, win);
43     MPI_Op_free(sum_op);
44
45     /* detach user buffer from window */
46     if (rank == 1)
47         MPIX_Am_win_buffer_detach(user_buf, win);
48     ...
49 }
```

Figure 6.5: Code example of AM

and RMA operations.

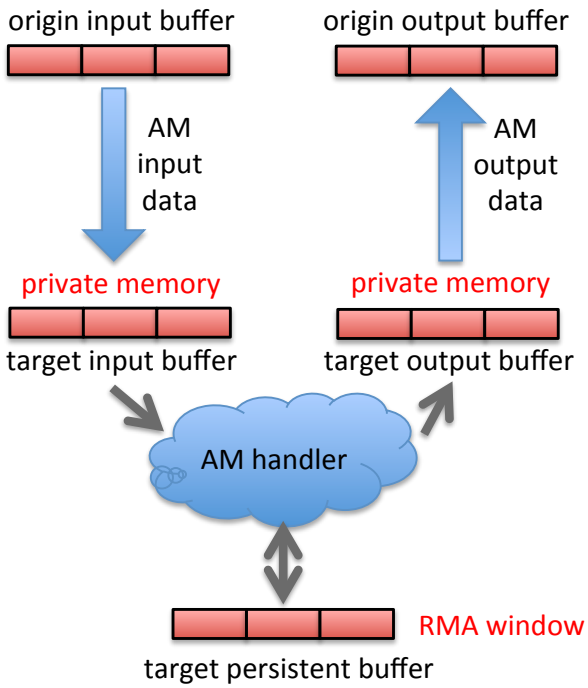


Figure 6.6: MPI-AM workflow

6.4.5 Correctness Semantics

In this section, we introduce important correctness semantics of MPI-interoperable AM framework, from perspective of ordering, concurrency, atomicity and memory consistency, to define how MPI-AM can work correctly and efficiently with other MPI messages within MPI infrastructure.

Memory Consistency. Two memory models are defined in MPI RMA specification: UNIFIED and SEPARATE. In the SEPARATE model, the MPI process can be viewed as having two copies of the window: one copy is the "public window", which is addressable by all processes, and another is the "private window", which is local to each process. In the UNIFIED model, there is a single copy of the window. In practice, the SEPARATE model is more natural for non-cache-coherent architectures in which the consistency of the

cache with respect to memory has to be handled in software by the MPI implementation, whereas the UNIFIED model is more natural for cache-coherent architectures in which such consistency needs to be managed properly in hardware. MPI RMA operations like MPI_PUT and MPI_GET access the public window copy, and local loads and stores access the private window copy. One primary difference between AMs and traditional RMA operations is that AM handlers access the private window copy instead of the public window copy. This is because operations involved in the AM function are local loads and stores invoked by the target process instead of MPI_PUTs, MPI_GETs and MPI_ACCUMULATEs invoked by the remote process. Furthermore, concurrent AMs at the same target process have a separate private window copy.

This difference between AMs and traditional RMA opertions brings several subtle interoperability issues. For instance, in the SEPARATE model, if an AM and a regular RMA operation update the same window within the same epoch, the state of the data in that window is undefined, even if these operations update nonoverlapping memory regions. This is because during the execution of an AM handler, if the target process fetches a block of data from cache and an RMA operation updates another nonoverlapping variable on the same cache line, such modification would be lost when the cache line is written back to memory. MPI runtime is unable to keep track of such accesses. Similarly, if two concurrent AMs modify non-overlapping regions on the same target window region, they both might have two different copies of the private window, and can therefore overwrite each other's modifications on that window memory. Furthermore, in both memory models, each AM handler has to gurantee that it uses the latest modifications by previous RMA operations and leaves the window in a consistent state for future following RMA operations. Since the AM handler function is performed upon the private window memory region, in the SEPARATE model, the MPI implementation will have to flush the cache back to memory before sending the AM completion notification to the origin process. In the UNIFIED model, even though the status of the cache is managed by hardware and coherency is ensured, the

MPI implementation still needs to perform a full memory barrier before and after the AM function handler, in order to ensure that future reads from the window memory sees the latest updated data.

Ordering. In MPI-AM, we define three types of ordering: (a) ordering between AMs with the same operation; (b) ordering between AMs with different operations; and (c) ordering between segments within one AM. By default, MPI-AM imposes strict ordering for all three types from the same origin process to the same target process on the same window with overlapping target memory region. For all other cases, there is no ordering requirements. The default strict ordering permits applications to reason about the state of the target window buffer when multiple AMs update it concurrently. The application knows that a later AM is guaranteed to see the modifications made by all previous AMs. However, such strict ordering requirements also place a significant performance overhead within the MPI implementation. For instance, if an AM is blocked because of lack of sufficient buffer space or any other reasons, a future AM might also need to block. To address this issue, we allow the user to release the ordering constraint of AMs using the MPI info hint am_ordering, which is set on the MPI window during window creation phase. This value is a comma-separated list of the required ordering with permitted values: sameop, diffop and sameam, for the three cited types of ordering, respectively. The default value for am_ordering is {sameop, diffop, sameam}, which imposes all three strict orderings. Any subset of those three orderings, or a value "none" can be passed to relax strict orderings of MPI-AM. Reduced ordering ensures can be beneficial for some applications, for example those which use AMs that only read the target data but do not modify it. For such kind of applications, the more relaxed semantics can give the MPI implementation freedom to reorder AM operations in order to achieve better performance. Note that AMs are completely unordered relative to other MPI operations.

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

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

Note that an MPI implementation might not be able to control the ordering of concurrent AMs. Thus, if the user requests strict ordering but provides an assertion to specify that

the AMs are safe to release the concurrency, an MPI implementation might still have to disable concurrency to satisfy the ordering requirement. We also note that concurrency of AMs does not restrict AMs from executing concurrently with other RMA operations.

Atomicity. MPI-AM framework does not guarantee atomicity of modifications between concurrent AMs or between AMs and other RMA operations. Thus, if two AMs update the same memory location on a target window, the resulting value is undefined. Similarly, if an AM accesses the same memory region as another RMA operation, the resulting value is also undefined. An exception is read-only accesses; if multiple AMs read from the same memory location, or if an AM and another RMA operation read from the same memory location, such accesses would return the actual value at the corresponding target memory. Note that the atomicity requirements are valid only for the public persistent buffer and do not impact the input buffer and output buffer that are private to each AM.

Other Considerations. One additional aspect is whether an AM handler can make a call of other MPI functions and how. Two factors must be considered carefully. First, an AM handler might be executed by the MPI implementation while it is making progress for another MPI function for instance, while it is waiting for a request to complete, in this situation, allowing the AM handler to execute an MPI function would result in the execution re-entering the MPI stack, thus requiring the MPI routines to be re-entrant safe. Secondly, since an AM might execute concurrently with the main thread which performs the application, calling MPI routines within an AM would require MPI to be initialized in a thread-safe pattern. Since the application does not know whether the MPI runtime would execute the AM concurrently with the main thread, the application would need to initialize MPI with a higher thread-level than it would otherwise require. To avoid these issues, in our current model we prohibit AM handlers to call other MPI functions.

6.5 Experimental Evaluation

In this section, we demonstrate the experimental results for the generalized MPI-interoperable AM framework. Results in Section 6.5.1 are gathered on a 310-node system, with each compute node containing 16 cores Total number of cores is 4,096. The nodes are connected with QLogic QDR InfiniBand Interconnect with fat-tree topology. Results in Section 6.5.2 are gathered on a 320-node system, each has two Intel Xeon X5550 quad-core CPUs, and QDR InfiniBand HCAs. Our implementation is based on MPICH (version 3.0.2). We implemented two common operations using AMs. The first operation is a *remote search* operation, where the origin issues muliple AMs with DNA sequences composed of 20 characters to the target process in order to search for matching DNA strings and return them to the origin side. Each segment has 20 characters as input and 20 characters as output. The second operation is a remote compute of the summation of *absolute values* of two arrays. Each segment has 100 integers as input and 100 integers as output. All experiments use an internal system buffer of 8 KB per peer process, except for Figures 6.8 and Figure 6.9 which vary the internal buffer size.

6.5.1 Microbenchmarks

In this section, we first illustrates the experimental results using multiple microbenchmarks, including measuring impact of message streaming, data buffering management and different configurations for semantic choices including concurrency and ordering of AMs.

Streaming Active Messages. As discussed in Section 6.4, MPIX_AM is designed to allow the MPI implementation to devide an AM into multiple smaller segments for better pipelining or to limit memory usage for temporary buffers. Here, we analyze the impact of such streaming / pipelining of segments within an AM. Figure 6.7(a) illustrates an experiment with the *remote search* operation, in which we measure the latency of a single

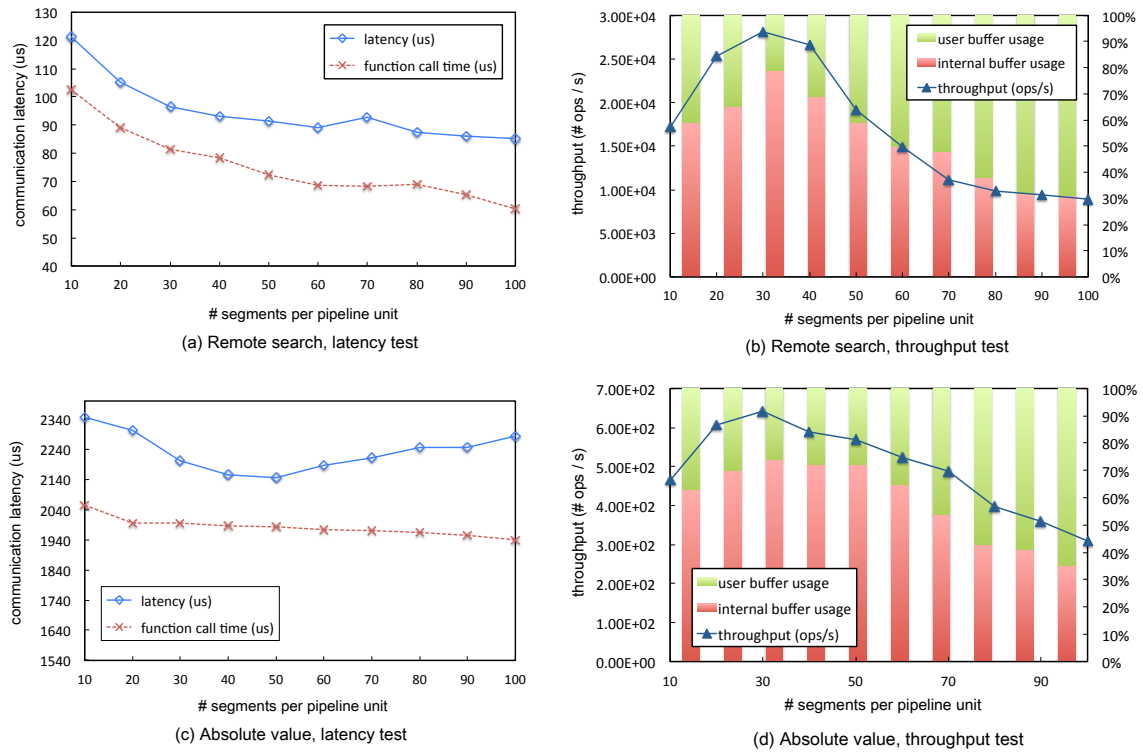


Figure 6.7: Communication latency and operation throughput with different numbers of segments per AM packet

AM within an epoch. When the epoch is closing, the AM has completed and a completion notification has been sent from target process to the origin process. Each AM consists of 100 segments, each segment requiring 20 bytes for each of the input and output data buffers. The MPI runtime uses an internal buffer of 8 KB per peer process; thus, with a single AM within the epoch, we are guaranteed that the MPI internal buffer can serve the entire AM and the user buffer is never being used.

We can observe that there is a continuous drop in latency as we increase the size of each pipeline unit. With a pipeline unit of 10 segments the latency of each AM is 115μs, while with a pipeline unit of 20 segments the latency drops to 95μs, about 17% performance degradation. When we further increase the pipeline unit, the latency is dropped by another 16%. To analyze this behavior, we profiled the execution time by measuring the cost of computation without the overhead of AM data transfers and synchronization. The time drop in computation time with increasing pipeline unit size is because of the reduction

in the total number of function calls. When each pipeline unit is 10 segments long, the AM handler is invoked 10 times in total, each is involved with 10 segments to perform computation. However, when each pipeline unit is 100 segments long, the AM handler is invoked just once with 100 segments to perform computation. While the total amount of computation in both cases is the equivalent, the former case has a 10X larger times of function invocations. We also observe that the time to execute the computation closely follows the trend of the AM latency. The additional overhead compared to the computation time is attributed to the AM data transfer and synchronization overhead.

In Figure 6.7(b) we illustrates a similar experiment with the *remote search* operation, where we measure the throughput by performing 100,000 AMs within an RMA epoch. Each AM consists of 100 segments and each segment requires 20 bytes for each of the input and output buffers. The MPI implementation uses an internal system buffer of 8 KB per peer process; thus AMs internally use the system internal buffers when it is available and fall back to the user buffer when internal buffers are all used up. We can seen that when the pipeline unit is 40 segments, the highest throughput can be achieved. To analyze this observation, we profiled the number of segments that utilize the MPI system internal buffers and the number of segments that use the user buffers in Figure 6.7(b). As we increase the pipeline unit, there is an increasingly large fraction of the segments that uses the user buffer, thus resulting in additional more synchronization with the target side. At the same time, when pipeline unit size is very small, the number of function calls can add a quite high overhead, as illustrated in Figure 6.7(a). Consequently, we expect the best throughput can be achieved at somewhere in between. For the *remote search* operation, a pipeline unit of 40 segments happened to be the best case.

Figures 6.7(c) and Figure 6.7 (d) show the results of experiments similar to those above but for the second type of computation: *absolute values* computation. The performance trends are similar to Figures 6.7(a) and Figure 6.7(b), except that in Figure 6.7(c) the AM latency reaches its lowest point at a pipeline size of 40 segments and increases after that.

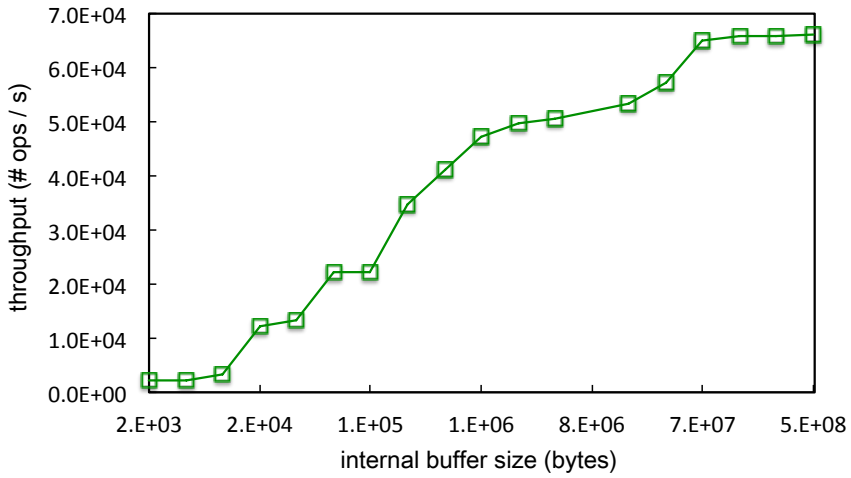


Figure 6.8: Throughput: impact of system buffer size

This is because unlike the *remote search* computation, the *absolute values* computation is more input data intensive, which requires a larger amount of data to be transferred between the origin process and target process. Thus, the pipeline size makes a obvious difference in the overlapping performance of computation and data movement.

Impact of Internal Buffers. As we described in the design section, the user is required to attach enough temporary buffers that can be used by the MPI implementation to stage AM input and output data. However, the MPI implementation can always choose to allocate additional internal buffers in order to improve performance if it has available internal buffers. In our implementation, the target process pre-allocates a small amount of buffer space and statically assign them to each origin process. Total internal buffer size would be this size times the number of processes. In this section, we evaluate the impact of such internal temporary system buffers on the performance of MPI-AMs.

Figure 6.8 shows the performance impact of increasing system internal buffer size on the throughput of AMs. We used the *remote search* benchmark for this experiments, in which each AM contains 100 segments and each epoch issuing 100,000 AMs. We observe that the throughput of AMs increases with growing system internal buffer size up to a point and

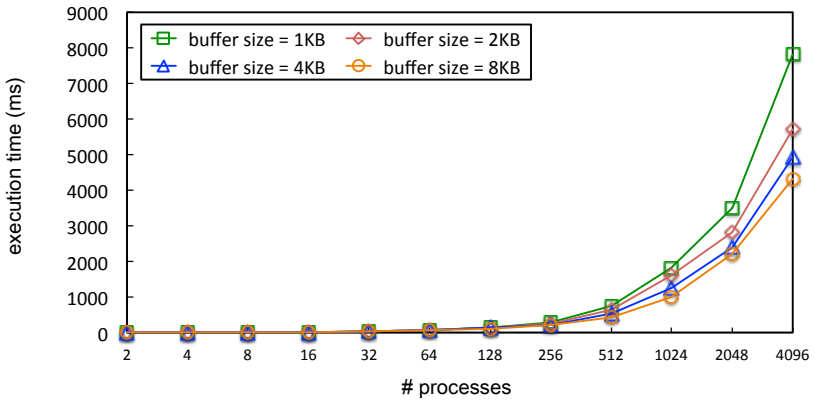


Figure 6.9: Execution time: impact of system buffer size

then keeps flat. The reason for the performance increase is quite straightforward: when more internal system buffer space is available, the MPI implementation can directly use the system internal buffer instead of performing a handshake synchronization with the target process to get access to the shared user buffer space. With growing internal buffer space, the number of times such a handshake synchronization occurs is decreasing, which improves performance. The reason for the leveling off of the performance at large internal buffer sizes is relatively subtle: the amount of internal buffer size that a series of AMs can use is very limited. When the MPI runtime keeps issuing more AMs, previously issued AMs complete their execution on target side and frees up more buffers. Even with a large number of AMs, the runtime system can reach a steady-state where the data transmission and AM computation cound match up with each other and more temporary buffer space will not make a difference after certain point.

Figure 6.9 illustrates the performance impact of the internal system buffer size when multiple origins issue AMs to the same target process. This experiment is similar to the previous experiment except that a large number of origin processes issue AMs to the same target concurrently. We notice that for a small number of origin processes, the internal system buffer size does not make obvious difference. However, as the number of processes increases, there is as much as a 1.7 times performance difference.

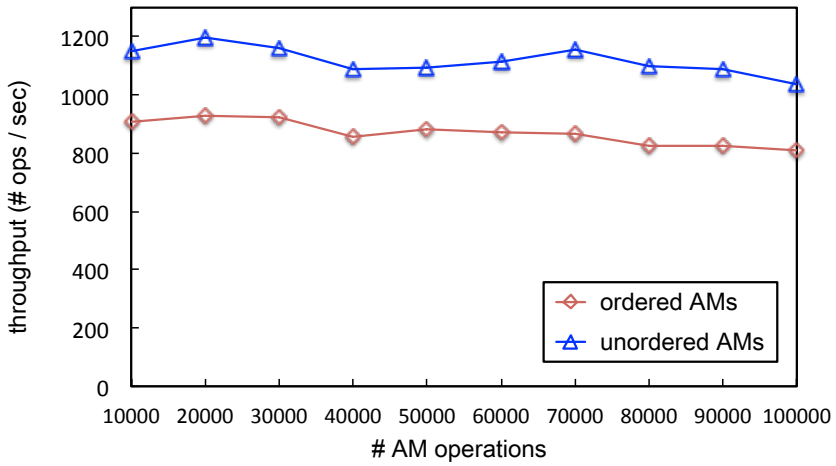


Figure 6.10: Throughput: impact of ordering

Concurrent Active Messages. In this section, we analyze the impact of whether executing AMs concurrently or not within MPI runtime. In our experiment, we have a number of origins issuing AMs to a same target. When there is no concurrency in the AMs by default, and all AMs are forwarded to the target and executed on target side. However, when the user turns on AM concurrency, each origin can take advantage of the fact that the window data is shared across processes within one node and each of them can compute directly at the origin process side. Figure 6.11 illustrates the performance through concurrency of AMs on an 16-core system. Without concurrency, when number of processes is 16, the aggregated throughput achieved by the AMs can be more than 9X worse than with concurrency enabled.

Impact of Ordering. In this section, we measure the impact of ordering between AMs within MPI runtime. In our experiment, we issue multiple AMs within a single epoch alternating between ones that requires a large temporary buffer and ones that need a small temporary buffer. When the user requires strict ordering among all AMs, the MPI implementation must finish all previous AMs before issuing the next AM. If an AM is blocked waiting for more memory space than what the MPI internal buffer can offer (thus requiring the support from the user buffer), all future AMs will also be blocked even though

they may fit into the MPI internal buffers. When the user release such ordering among AMs, runtime is free to issue later small AMs early, while waiting for the larger user buffer to be available for previous large AMs. Figure 6.10 illustrates the performance comparison by such lack of ordering. Removing strict ordering requirements can provide close to 25% performance improvement. On architectures such as Blue Gene machine, where multiple routes exist between the origin process and target process, the ability to issue AMs out of order can allow the MPI implementation to use multiple paths for operations between two processes, therefore brings further improving performance.

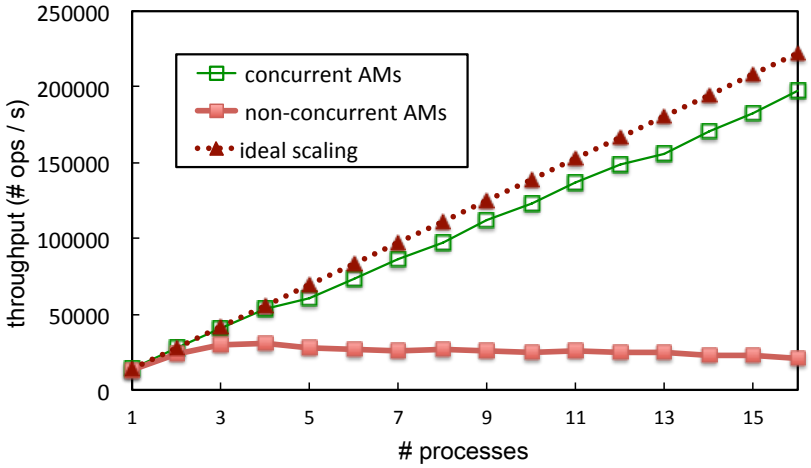
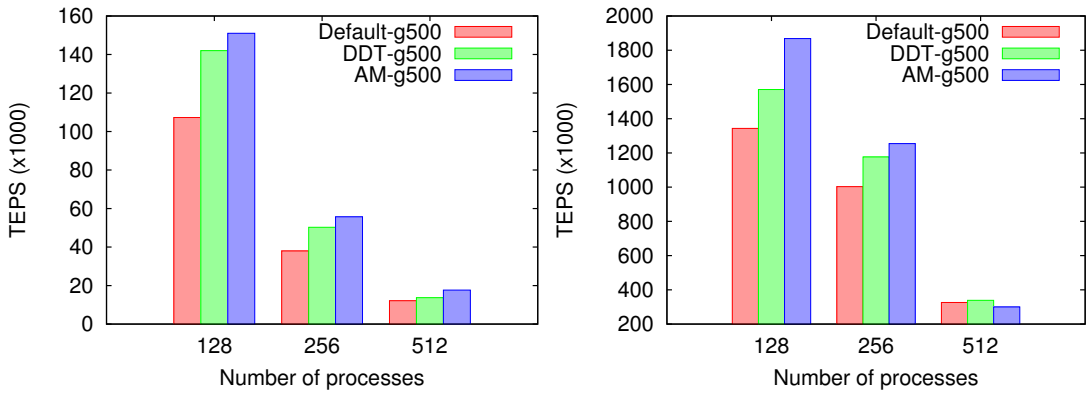


Figure 6.11: Throughput: Impact of concurrency

6.5.2 Graph 500 Benchmark

Graph 500 [11] is a relatively new benchmark used to test data-intensiveness of the system. It performs a breadth-first search (BFS) 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 operations among all the processes during FENCE epochs. A straightforward optimization improvement in the application is to combine a certain number of MPI_ACCUMULATE operations into a large MPI_ACCUMULATE operation with derived datatypes. Such optimization reduces the large number of small communication transactions



(a) Small problem size (2^{15} vertices) (b) Large problem size (2^{20} vertices)

Figure 6.12: Graph 500 comparative performance results

towards each target process. Such an improvement can simply be implemented using MPI derived-datatypes to send coalesced data resulting from the local accumulation of the data for each target in each FENCE epoch. However, the derived datatypes approach must follow the non-overlapping constraint required by the MPI-3 specification for target datatypes in MPI_ACCUMULATE operations. By using AM, however, which uses user-defined functions in MPIX_AM, the above constraint from derived datatypes can be completely avoided. Before issuing out operations to each target at the end of the epoch, the AM approach combines the locally accumulated data into large sparse arrays. This computation, which occurs once for each target in each FENCE epoch, is less computation intensive compared to the non-overlapping constraint checking that the derived datatype approach requires for each local accumulation.

In Figure 6.12, we illustrate the evaluation results of TEPS to compare the default one-sided implementation (Default-g500) with a derived-datatype-based implementation (DDT-g500) and our AM-based implementation (AM-g500). The tests are performed for 2^{15} vertices and 2^{20} vertices respectively over 128, 256 and 512 processes.

In Figure 6.12a, both DDT-g500 approach and AM-g500 approach perform better than Default-g500 approach, and AM-g500 approach performs even better than DDT-g500 approach. The same trend is observed in Figure 6.12b over 128 and 256 processes. However,

AM-g500 approach performs worse than DDT-g500 approach over 512 processes in Figure 6.12b. The case where AM-g500 approaches performs worse compared to DDT-g500 and Default-g500 is a consequence of several general behaviors of the MPI one-sided implementation happened in Graph 500 benchmark. For both DDT-g500 and AM-g500 approaches, the job size variations create a trade-off between the size of communication towards each target and the total number of targets each process talk with. The larger the problem size, the larger the number of target processes but the smaller the message sizes to each process. Consequently, the coalescing effect tends to be slowed down when the job size increases to larger scale. A conclusion can be reached where the overhead of each coalescing approach will not be offset enough by the communication overhead saved by coalescing the messages. The important observation, however, is that at a fixed scale, Graph 500 benchmarks remains scalable with respect to the number of processes until a peak is achieved, and after that TEPS enter a phase of performance down. The numbers of processes shown on Figure 6.12 happen to be in the range of performance drop for 2^{15} vertices and 2^{20} vertices. We confirm that 128 is the optimal job size for 2^{20} vertices for our test environment when job sizes are changed by a factor of 2. Therefore, no matter what implementation to run, an informed user is less likely to execute Graph 500 benchmark over job sizes where the AM-g500 approach performs worse.

6.6 Conclusion

In this chapter, we presented a design and implementation of a framework: MPI-interoperable generalized Active Messages. We first analyzed usage restrictions and performance disadvantages of one naive design: extending existing MPI_ACCUMULATE-like operations to support user-defined functions, those limitations includes data layouts, data access, and memory inefficiency issues. Based on the analysis, we proposed a new design for MPI-interoperable AMs, including various strategies to handle AMs within MPI runtime includ-

ing message streaming, buffer management, operation registration, correctness semantics, to provide a more flexible and general usage model. We evaluate the MPI-AM framework via a comprehensive microbenchmarks and Graph 500 kernel benchmark to demonstrate the performance impact of message streaming, message buffering management and different semantic choices.

CHAPTER 7

Optimization Strategies for MPI-Interoperable Active Messages

In Chapter 6, we proposed an MPI-interoperable Active Messages framework, which allows MPI-based applications to incrementally utilizing AM capabilities and avoid being rewritten. While we presented a baseline implementation of how AMs work with the existing MPI infrastructure in Chapter 6, it had several performance disadvantages. For instance, the semantics of the MPI-AM framework allows users to provide buffers on target to stage AM incoming data, and such buffer could be used by multiple different processes issuing AMs on a particular same target process. Because such user buffers are shared among all other processes, significant performance overhead was introduced in the synchronization phase, especially on large-scale machines. Furthermore, for a highly irregular application where the amount of data involved in each AM is quite variable, the previous baseline implementation can lead to a significant amount of additional unnecessary data being transferred, resulting in additional performance overhead.

In this chapter, we identify those performance disadvantages through a thorough analysis of the behavior of the MPI-AM infrastructure. Our analysis shows significant stalls and idleness during MPI RMA synchronization, which is increased as system size grows. Our analysis also shows that a significant amount of unnecessary data is transmitted for in highly irregular applications. To address all those limitations, we propose three optimization

strategies to improve the performance. The first strategy is an implicit optimization which can avoid additional internal synchronization and improve performance. The second and third strategies are more explicit methods. The second strategy uses an MPI hint to learn additional application information from user in order to further reduce synchronization overheads in certain application scenarios. The third strategy proposes a new "vector-based" AM interface which allows highly irregular applications to be able to better describe their irregular data layout and reduce the amount of data transferred in the application.

In this chapter, in addition to the detailed description of these optimization techniques, we also present a reference implementation and a thorough performance analysis of the optimization strategies on a 4096-core InfiniBand cluster. Our evaluation demonstrates a significant performance advantage from the optimization techniques.

7.1 Performance Shortcomings of MPI-Interoperable Active Messages

In this section, we analyze the performance limitations in the previous base implementation of MPI-interoperable AM framework. Based on this analysis, we propose three optimization strategies in Section 7.2.

7.1.1 Synchronization Stalls in Data Buffering

As described in Section 6.4.4, the semantics of MPI-AM framework require that it is user's responsibility to guarantee that there are enough temporary buffers provided on current window window to handle the input / output data corresponding to the AM can be accommodated at the target. However, the MPI implementation can provide additional internal buffers to improve performance if such buffers are available.

System internal buffers can be managed in multiple methods. For example, a large amount of memory can be shared among all processes. In such method, each origin process must

coordinate with the target to reserve portion of the buffer before it can issue an AM. Another possible method is to statically divide the buffer among the origin processes, so each origin process acquire exclusive access to its private system buffer. The advantage of this approach is that since the buffer associated with each origin is exclusive buffer, no additional synchronization is required between origin and target in order to use that internal buffer. The shortcoming, however, is that such static partitioning method reduces the amount of buffer available to each origin. Other design choices also exist where one target could choose to dynamically manage the amount of exclusive buffer space available to each origin process at runtime. In our implementation, we have not investigated all possible design choices. Instead, we choose the second method: statically partitioning the internal system buffer so that each origin process has exclusive access to a part of the buffer.

No matter which design is selected for the internal system buffer management, we note that those system buffers are limited. When each AM is large, or when a large number of AMs are issued out, the system buffer will eventually be used up, and the MPI implementation has to use the user-provided buffer. Given the shared nature of those user buffers, however, each origin must perform a "handshake" synchronization with the target process in order to reserve some user buffer before it can issue its AM.

The overall handshake protocol is shown in Figure 7.1. In this scenario, considering an AM that has a large amount of AM segments. The first few segments that can fit into the system internal buffers are issued out immediately. Once those system buffers are exhausted, however, the origin needs to send a handshake message in order to reserve space in the shared user buffer before it can issue the next AM segment, thus it needs to idly wait for buffer space at the target to become available. Depending on which fraction of time the origin spends for waiting, the performance of the MPI-AM can be substantially impacted. Issuing multiple nonblocking operations does not address this issue, since the origin process issue send only as much data as it knows the target can handle with.

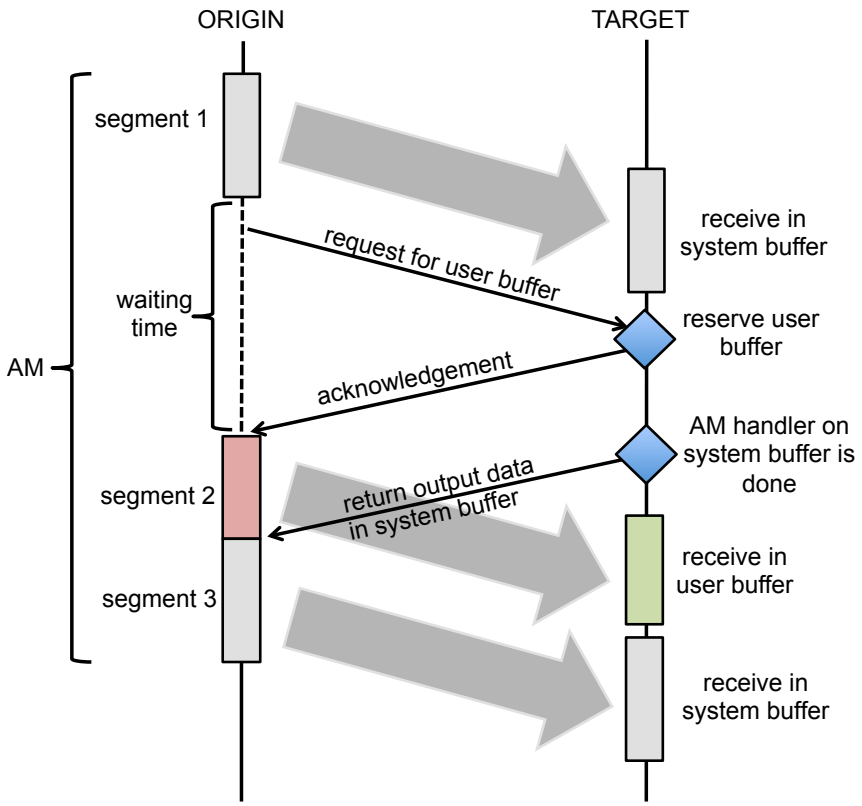


Figure 7.1: Handshake operation for reserving user buffers

7.1.2 Inefficiency in Data Transmission

A common characteristic of several irregular applications is that the amount of data returned by an AM is often data-dependent. For instance, in bioinformatics genome assembly applications such as Kiki, where an input DNA query is searched on remote datasets, the amount of data returned depends on how many DNA matches the AM handler can find in the remote dataset. Such information, unfortunately, cannot be predicted easily. Therefore, the semantic of MPI-interoperable AMs in such kind of applications requires the application to allocate a large local output buffer and issuing AMs that return data into this buffer.

Such a model has two obvious shortcomings. First, the amount of buffer allocated for output data can potentially be very large. Second, with the semantic of MPIX_AM, the AM handler cannot specify a different amount of output data size for each of those segments; thus, the amount of data returned to the origin process is equal to the total size of buffer space

allocated. It is obvious that such situation can be very wasteful in irregular applications where the amount of data returned can vary dramatically among AM segments.

7.2 Optimization Strategies

Based on the performance limitations described in Section 7.1, we present three optimization strategies for MPI-interoperable AM framework, which can improve data buffering management and efficiency in data transmission of MPI-AM.

7.2.1 Efficient Data Buffering Schemes

In this section, we describe two optimization strategies which can improve the efficiency of buffering management in MPI-AM framework.

Autodetected Exclusive User Buffers. The first optimization technique utilizes the application synchronization to reduce the internal synchronization required for reserving user buffers on target side. As specified in MPI-3, MPI RMA (or AM) operations can be issued only within an MPI RMA epoch, either in Passive Target mode or in Active Target mode. In Passive Target mode, origin process first issues an MPI_WIN_LOCK to target process, followed by multiple AMs, and then finishes the current epoch by calling MPI_WIN_UNLOCK. Such an epoch can be initiated in either MPI_LOCK_SHARED or MPI_LOCK_EXCLUSIVE mode. If an origin process acquires an "exclusive" lock at the target window, no other origin process can acquire either "exclusive" or "shared" lock at the same target on the same 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 and on the same window concurrently.

Our optimization strategy takes advantage of this model by making MPI runtime internally keep track of the lock acquisition states of each window. Thus, if an origin has acquired

an "exclusive" lock at a target window, it should be the only process that can access the target window and the attached user buffers. In such scenario, we need to send only one synchronization message right after MPI_WIN_LOCK call to get buffer information about the target user buffers. For all subsequent AM operations, no more synchronization messages are required. Note that this optimization is transparent to the user but is an implicit method.

Here we need to handle a corner case with respect to detachment of user buffers. MPI-based applications are allowed to attach / detach arbitrary number of user buffers to / from a window dynamically. Before they detach a user buffer, however, they should ensure that there is no AMs that are currently executing within the user buffer. Thus, an example such as the one shown in Figure 7.2 is a valid program. Note that in the program, the amount of user buffer space attached to the window has changed while process is in the exclusive lock epoch. That is, the AMs issued from lines 9–11 have access to both user_buf_1 and user_buf_2, which are attached to the target window. 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 situations, our optimization of querying for the available user buffer space just once at the beginning of the passive epoch would not be correct. To address this scenario properly, we need to re-synchronize the user buffer information after every synchronization call like MPI_WIN_FLUSH. Such re-synchronization, however, can result in performance overhead in situation where the user buffer was not detached at the target. Unfortunately, the MPI implementation cannot detect such situation automatically. To tackle this issue, we allow users to pass a MPI hint to the MPI implementation at window creation phase using MPI info key am_buf_interleave_am_detach. The default value of true indicates that the user can interleave AM operations with MPIX_AM_WIN_BUFFER_DETACH operations, thus requiring additional synchronization as mentioned above. By setting this value to false, however, the user ensures that MPIX_AM_WIN_BUFFER_DETACH operations will be never interleaved with any AM operations, thus requiring no other synchronization. In such cases, the additional synchronization operation in MPI_WIN_FLUSH can be avoided.

```
1       ...
2       if (myrank == 0)
3       {
4           /* Barrier to ensure that buffers are attached */
5           MPI_Barrier(MPI_COMM_WORLD);
6
7           MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
8
9           /* AMs should have access to both user buffers */
10          MPIX_Am(...);
11          MPIX_Am(...);
12          MPIX_Am(...);
13
14          MPI_Win_flush(1, win);
15
16          /* Barrier to inform that flush has completed */
17          MPI_Barrier(MPI_COMM_WORLD);
18          MPI_Barrier(MPI_COMM_WORLD);
19
20          /* AMs should have access to one user buffer */
21          MPIX_Am(...);
22          MPIX_Am(...);
23
24          MPI_Win_unlock(1, win);
25      }
26      else if (myrank == 1)
27      {
28          MPIX_Am_win_buffer_attach(user_buf_1, 100, win);
29          MPIX_Am_win_buffer_attach(user_buf_2, 200, win);
30
31          /* Barrier to ensure that buffers are attached */
32          MPI_Barrier(MPI_COMM_WORLD);
33
34          /* Barrier to inform that flush has completed */
35          MPI_Barrier(MPI_COMM_WORLD);
36
37          /* detach user buffer */
38          MPIX_Am_win_buffer_detach(user_buf_1, win);
39          MPI_Barrier(MPI_COMM_WORLD);
40      }
41      ...
```

Figure 7.2: Example of attaching / detaching of AM buffers

User-Defined Exclusive User Buffers. The second optimization we introduce is for situations where the MPI implementation cannot automatically detect exclusive user buffer access, such as with shared locks or in MPI Active Target synchronization mode. In some cases, the application can algorithmically decide the maximum size of target buffer that would be used by other origin processes and thus can decide the amount of target buffer available to a given origin process. In such cases, if the application can pass this information to the MPI implementation as an MPI info hint during window creation, the MPI implementation can use this information to potentially reduce synchronization overhead.

In our approach, we use a target-specific MPI info key (am_user_buf_<rank>); the info value specifies the size of the user buffer space on that target that is guaranteed to be available (in bytes). We emphasize that this value specifies only the guaranteed buffer space and therefore is conservative. More user buffer space might be dynamically available to the MPI application, which it can query for, using handshake protocol between origin and target. For AMs that fit in the "exclusive user buffer" space, however, no further handshake synchronization is needed.

7.2.2 Improving Efficiency in Data Transmission

As described in Section 7.1.2, with previously proposed MPIX_AM, the AM handler cannot specify a different amount of output data size for each segment in that AM. Thus, a fixed amount of output data, which equals to the maximum AM output size, is returned to the origin process when AM is completed. In this section, we propose effective strategies to improve the efficiency of data transmission in such situation.

Vector-Based Interface. We first propose a new function for vector-based AMs, called MPIX_AMV, and an associated AM handler prototype, called MPIX_AMV_USER_FUNCTION,

```
MPIX_AMV_USER_FUNCTION (input_addr, input_segment_count,
                        input_segment_datatype, persistent_addr,
                        persistent_count, persistent_datatype,
                        output_addr, output_segment_count,
                        output_segment_datatype, num_segments,
                        segment_offset, output_segment_counts[])
  IN      input_addr               address of input buffer (choice)
  IN      input_segment_count      number of entries in each segment
                                   in input buffer (non-negative integer)
  IN      input_segment_datatype   datatype of each entry in input
                                   buffer (handle)
  INOUT   persistent_addr          address of persistent buffer (choice)
  INOUT   persistent_count         number of entries in persistent buffer
                                   (non-negative integer)
  INOUT   persistent_datatype      datatype of each entry in persistent
                                   buffer (handle)
  OUT     output_addr              address of output buffer (choice)
  OUT     output_segment_count     maximum number of entries in each
                                   segment in output buffer (non-negative
                                   integer)
  OUT     output_segment_datatype  datatype of each entry in output
                                   buffer (handle)
  IN      num_segments             number of segments in input and
                                   output buffers (non-negative integer)
  IN      segment_offset           current segment offset in input and
                                   output buffers (non-negative integer)
  OUT     output_segment_counts[]  actual counts of entries in each
                                   segment in output buffer (non-negative
                                   integer array)
```

Figure 7.3: Prototype of vector-based AM user-defined function

as illustrated in Figure 7.3 and Figure 7.4. Those functions are "vector" versions of the original MPIX_AM_USER_FUNCTION and MPIX_AM functions proposed in Section 6.4.3. There is one new vector argument, output_segment_counts, is added to these new APIs. This argument is an integer array with length of num_segments, each entry indicates the count of elements in the corresponding output segment. For the AM handler, the MPI runtime allocates the output_segment_counts array beforehand and passes to the handler, but how much data is actually generated needs to be filled by the handler function itself.

Output Data Layout. One design choice associated with those vector-based AMs is how much buffer space has to be allocated for the origin output buffer and how data should be laid out in this buffer. Since the amount of data that will be generated within the AM is unknown, the user cannot know the required buffer space required. Thus, we still allocate the origin output buffer to be large enough which can fit the maximum data size returned by the current AM handler.

Regarding to the data layout of the origin output buffer, however, the most intuitive approach would be to put the entire output data in a contiguous segment of the output buffer. While it is a convenient API for the user to use, such a data layout has several performance limitations, especially with respect to out-of-order execution of AM segments, as shown in Figure 7.5. For instance, suppose the AM has four segments. As shown in Figure 7.5(a), if the latter two segments in the AM are got executed and returnd earlier, the origin process cannot know at where it needs to place those output data, since it does not know how much output will be returned by the first two segments. In such cases, the MPI implementation has to either buffer those out-of-order data or put data in the user buffer and reorder it once all of the data is returned back. Both strategies are expensive and bring performance overhead. On the other hand, to prevent such complexity, users can impose strict ordering among AMs to prevent out-of-order execution, but that will sacrifice the concurrency of out-of-order AMs and the performance improvement.

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

Figure 7.4: Prototype of vector-based AM trigger routine

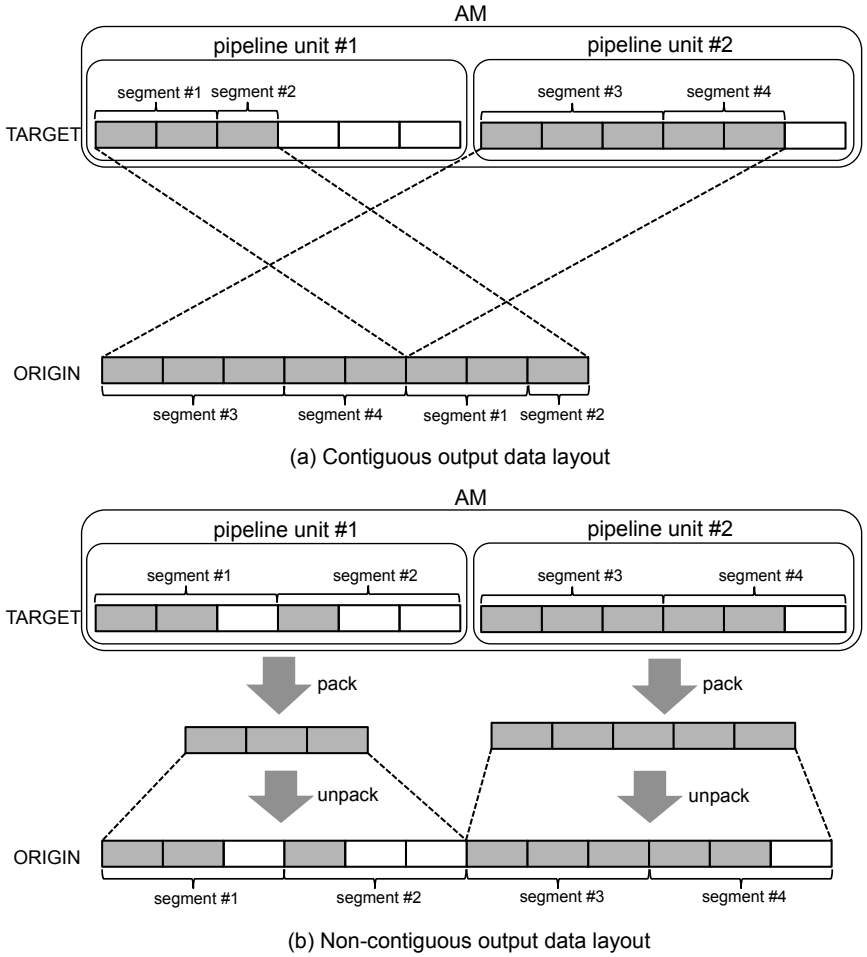


Figure 7.5: Different strategies of origin output data layout

An alternative model is that, the output data is not placed in a contiguous buffer, but each segment is put at a fixed offset calculated based on the maximum output size that each segment can generate, like what is illustrated in Figure 7.5(b). Such model may be slightly more inconvenient for user to use, however, the performance improvement of this approach is much higher. In particular, since the location of output data in each segment is predetermined, no additional buffering or reordering process is required. Data can be put at the correct location as soon as it arrives on the origin side. For our framework, we chose this approach.

Data Packing vs. Data Transmission. In MPIX_AMV, because each segment

may return an uneven amount of data, the output data can be noncontiguous in memory both at the target side within the AM handler and at the origin side after data is returned. Therefore, the MPI implementation at the target side needs to consolidate this data into a temporary packing buffer in order to transfer it to the origin process, which in turn will unpack the data onto the origin output buffer. In contrast, with MPIX_AM, due to the fact that the amount of data generated is predetermined, the communication is often from contiguous memory to contiguous memory.

The difference between the packing strategy in MPIX_AMV and the complete data movement strategy in MPIX_AM can be significant, in favor of MPIX_AMV, when the amount of actual data generated is much less 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 unnecessary data. However, as the amount of data generated by the AM handler increases as percentage of the maximum buffer size, this difference will reduce, and the packing overhead starts to dominate in the performance. To address this issue, in our framework, we internally maintains a system-specific threshold to decide when to pack data and when to give up packing but just transmit the entire output data. When the amount of output data is below this threshold, we pack and transmit the packed data. When the output data is above this threshold, we transmit all the data but give up packing optimization, therefore the transmitted data includes garbage data in the buffer that was not generated by the AM handler.

We emphasize that, in both strategies, the entire count array of the output lengths is also transferred to the origin process. Thus the total data transmitted will be slightly more than what MPIX_AM transmits when the handler generates the maximum amount of data. In such situation, MPIX_AM would be a better choice.

7.3 Experimental Evaluation

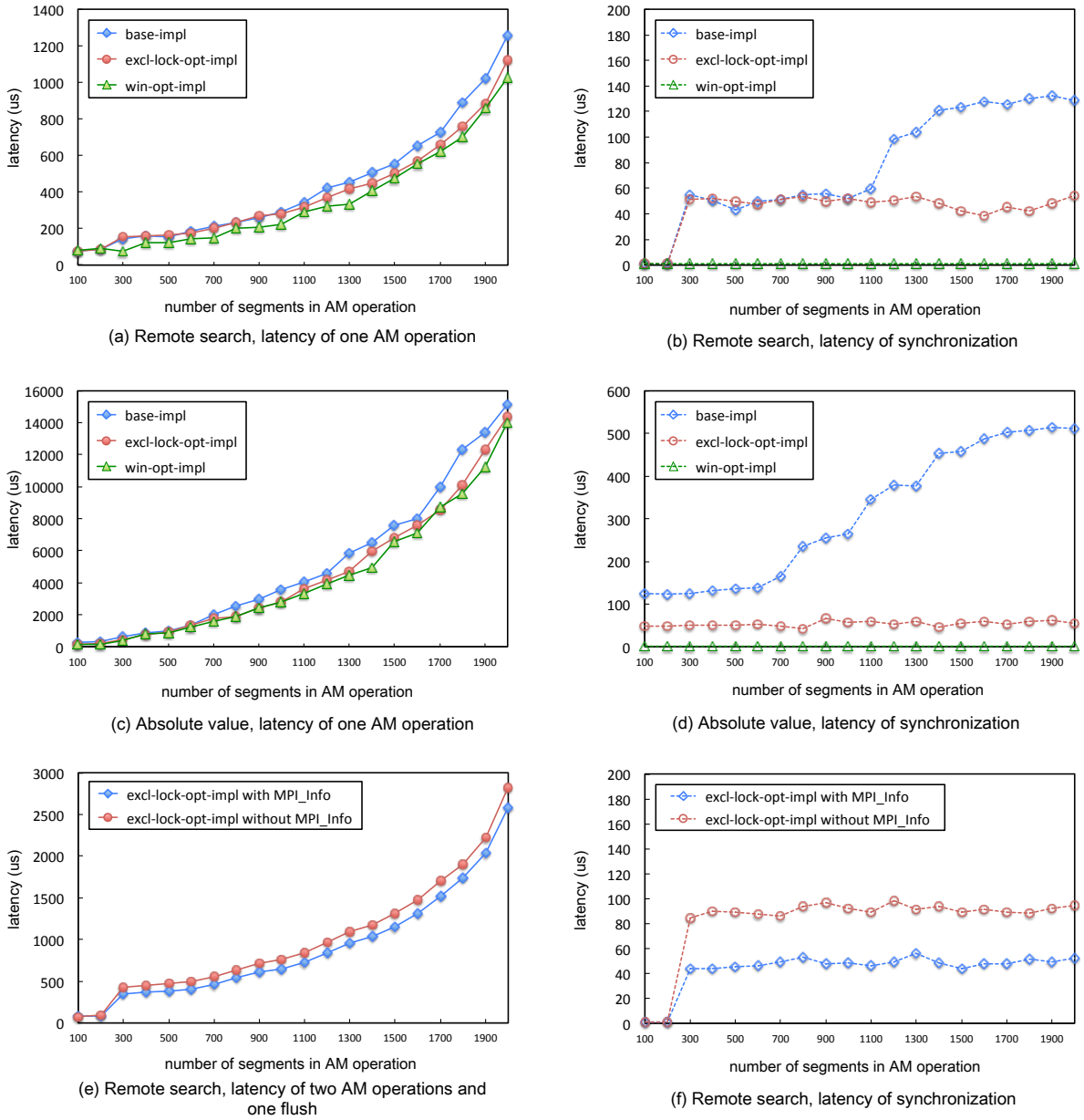
In this section, we present our evaluation using a 310-node system, with each compute node consisting of 16 cores. Totally there are 4,960 cores. The nodes are connected with Intel / QLogic QDR InfiniBand. Our implementation is based on MPICH (3.1b1). We implement two kinds of AM operations. The first one is a *remote search* operation, where the origin issues AMs with DNA sequences to search for matched DNA sequences in a remote dataset and return matches to the origin process. This is the most common operation used in genome assembly applications such as Kiki and SWAP [1]. Each segment consists of 20 characters (1 sequence) as input and 20 to 200 characters (1 to 10 sequences) as output (experiments in Section 7.3.1 return 1 sequence per segment, and experiments in Section 7.3.2 return multiple sequences per segment). The second operation is a remote computation of the summation of *absolute values* of two arrays. In this test, 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 7.6 and Figure 7.9, each process attaches 32 MB of user buffer.

In Section 7.3.1, we compare the performance of the first two optimization approaches. We use *Excl-Lock-Opt-Impl* for autodetected exclusive user buffers and *Win-Opt-Impl* for user-hinted exclusive user buffers, and we compare both of them with a base implementation that does not take advantage of either optimization (*Base-Impl*). In Section 7.3.2, we compare the performance of MPIX_AM with vector-based AM, MPIX_AMV.

7.3.1 Effect of Exclusive User Buffers

In this section, we focus on four effects: communication latency, operation throughput, scalability performance, and network contention.

Communication Latency. In Figure 7.6, we present the latency of a single AM



(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

Figure 7.6: Communication latency

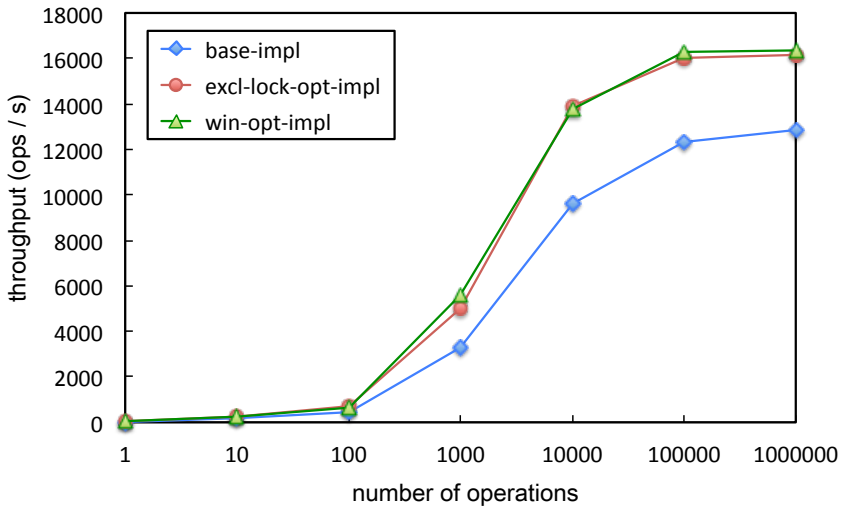


Figure 7.7: Operation throughput for *remote search*

with a *remote search* operation. There are two processes 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 7.6(a) that *Excl-Lock-Opt-Impl* optimization reduces the latency by around 10% compared with the *Base-Impl* and that *Win-Opt-Impl* optimization can further reduce latency by another 10%. We analyze those results by evaluating the time spent on synchronization in Figure 7.6(b). The figure shows that *Win-Opt-Impl* optimization spends no time on synchronization. This is 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, in *Excl-Lock-Opt-Impl* optimization, runtime does spend some time on synchronization messages, but the time spent does not increase with message size; in comparison, the synchronization time spent by *Base-Impl* increases with message size. This is because *Excl-Lock-Opt-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 *Base-Impl* does not utilize any hints on exclusivity, however, it needs to wait for previous AM segments to complete execution and reserve new buffers for the rest. We emphasize that both *Base-Impl* and *Excl-Lock-Opt-Impl*

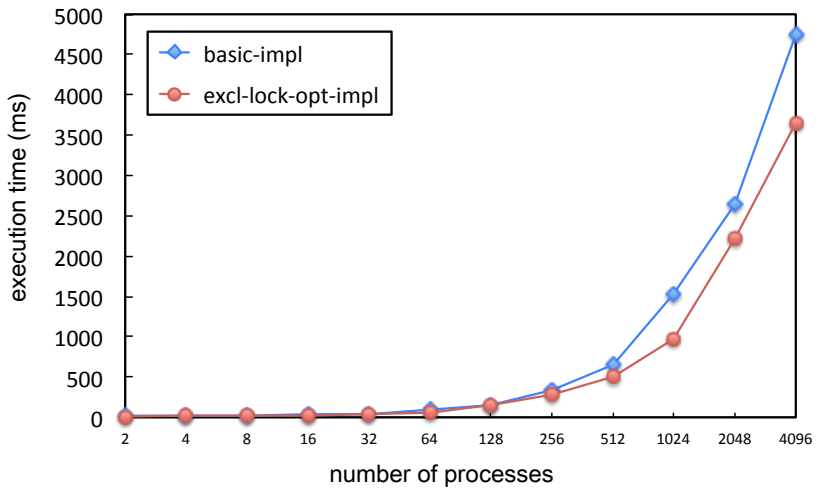


Figure 7.8: Scalability performance for *remote search*

optimization spend zero time in synchronizing with target process when the AM data is less than 200 segments. This is because in these cases the AM data is small enough to fit into the system buffer on the target.

Figures 7.6(c) and Figure 7.6(d) show a similar trend, but for *absolute values* operation. We observe that all three implementations perform similarly when the message has small size. As the message size grows, the two optimized implementations outperform *Base-Impl*.

In Figure 7.6(e), we evaluate the latency of two AM operations with one MPI_WIN_FLUSH between them, running with and without any user hint to specify whether user buffers are detached at the target during the MPI_WIN_FLUSH. As illustrated in the figure, the MPI info hint allows communication latency to be improved by around 10%. We further analyze this overhead by measuring time spent in the synchronization. As shown in Figure 7.6(f), the synchronization time is consistently reduced by 50%. This is because the MPI info hint allows the MPI implementation to have just one handshake instead of two handshake operations when there is no MPI info hint provided from the user. We also analyzed results from *absolute values* benchmark and it has similar performance.

Operation Throughput. In Figure 7.7, we measure the performance of operation

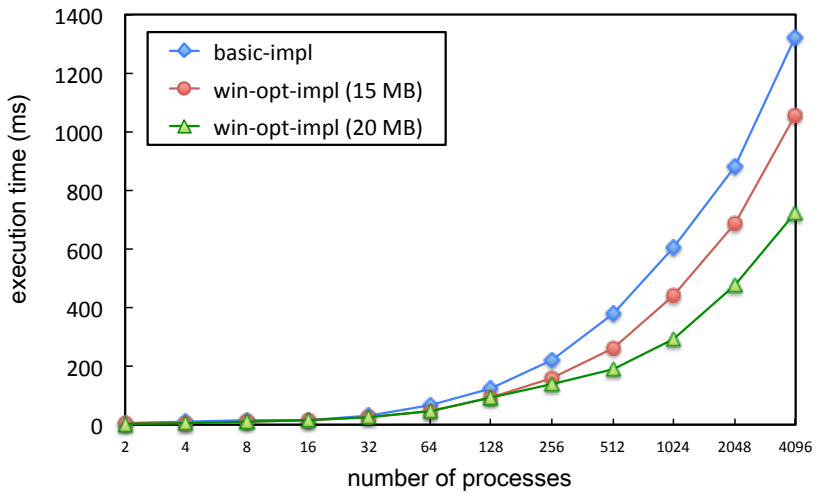


Figure 7.9: Contention performance for *remote search*

throughput when the origin process issues 1 to 100,000 AMs during the RMA epoch. The figure illustrates that both *Excl-Lock-Opt-Impl* and *Win-Opt-Impl* optimizations can achieve around 25% and 30% improvement compared with *Base-Impl*. This is expected because both the optimized versions have much smaller synchronization overhead. On the other hand, in *Base-Impl*, the origin process has to issue at least one synchronization message before each AM and wait for its response message before issuing that AM.

Network Contention. In Figure 7.9, we presents the benefit of *Win-Opt-Impl* optimization when we increase network contention situation. In this test, every four origins share a target, which is not an immediate neighbor of any origin process. Each origin issues a number of AMs to the same target. The experiment is set up such that on each origin process, all AMs will consume at most 20 MB of the temporary buffer at the target side. Therefore, we attach totally 80 MB of user buffer on the target.

We perform two experiments with MPI-AM framework. In the first test, we provide a MPI hint of "20 MB" to each origin process. In this test, because all the AMs together can consume at most 20 MB of buffer, no additional synchronization is needed by the MPI implementation. In the second test, we provided a MPI hint of "15 MB" to each origin.

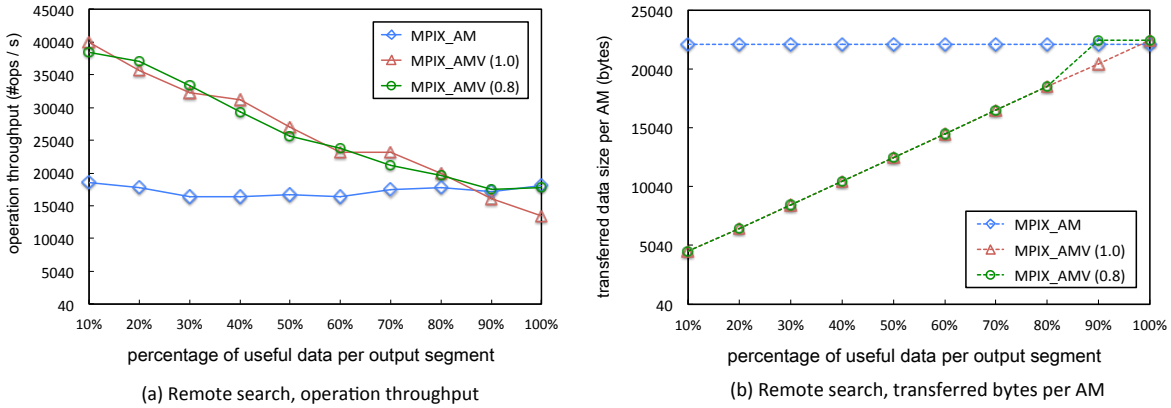


Figure 7.10: Operation throughput of MPIX_AM and MPIX_AMV

In this situation, some AMs can be triggered without having to coordinate with the target process, but such coordination is required for the other AMs. From the evaluation results, we can observe that with a MPI hint of "15 MB," the performance can be improved by 20% at scale. With a MPI hint of "20 MB," the performance can be improved by 50% at scale.

Scalability Performance. Figure 7.8 shows the benefit of *Excl-Lock-Opt-Impl* optimization when a large amount of origin processes compete for an exclusive lock at the same target concurrently. We run this experiment with an increasing number of processes. We can observe that, at 4,096 processes, *Excl-Lock-Opt-Impl* optimization can achieve 20% improvement in performance due to the reduced synchronization overhead.

7.3.2 Comparison between MPIX_AM and MPIX_AMV

In this section, we measure the throughput of the *remote search* operation using three different AM trigger interfaces: (a) MPIX_AM, in which no packing / unpacking is performed and full output segments are returned; (b) MPIX_AMV (1.0), which is the vector-based AM trigger where the MPI implementation always performs packing / unpacking and returns packed segments; and (c) MPIX_AMV (0.8), which is 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 7.10(a) illustrates the operation throughput achieved when the total amount of generated output 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 unnecessary data transmitted between origin and target processes. The operation throughput of vector-based AM keeps decreasing, however, as the percentage of data generated increases. When the percentage is above 80%, there is no advantage in those two. Furthermore, MPIX_AMV (1.0), which performs packing / unpacking of data, is around 30% worse than MPIX_AM when the generated data approaches 100%. MPIX_AMV (0.8), however, switches to sending all the data when the generated data is more than 80%, so its performance degradation is limited to 10%. The overhead of the vector-based AM triggers comes from two reasons: (1) overhead of runtime packing / unpacking and (2) transmitting additional counts array to the origin. To analyze those two aspects, we plot in Figure 7.10(b) the actual data bytes transmitted between origin and target processes. The figure shows that at 100%, both MPIX_AMV (0.8) and MPIX_AMV (1.0) transmit more data than MPIX_AM, and such difference coming from the additional counts array that the vector operations need to transmit.

7.4 Conclusion

In this chapter, we analyzed the performance limitations in the generalized MPI-interoperable AMs presented in Chapter 6, 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 various performance evaluations.

CHAPTER 8

Asynchronous Processing of MPI-Interoperable Active Messages

As we introduced in Section 6.1, Active Messages are particularly suitable for data-intensive and irregular applications with irregular communication patterns, such as graph algorithms or bioinformatics applications. In such applications, the receiver may not know how many messages it needs to receive or who is the origin process.Because Active Messages paradigm does not require the receiver to post a receive to explicitly receive a message, the receiver does not need to know the communication pattern beforehand.

Because the application at the receiver does not need to invoke a routine in order to process the incoming active message, the parallel communication library must be prepared to internally process the incoming message as soon as it arrives the target. The parallel communication library can be implemented as checking for incoming messages only when the application invokes a communication routine, which means that if the application does not invoke a communication routine for a period of time, for example when the application is in a long computation loop, incoming messages will not be processed during that time. Because of this, asynchronous processing of messages is critical for the performance of Active Messages. Asynchronous message processing can be implemented by using a separate thread that would handle incoming messages and process the computation. Because this thread is dedicated to receiving and processing any Active Messages, the messages can be processed

immediately no matter what the application is currently doing.

However, adding an additional thread introduces overhead because runtime needs mutexes to synchronize among threads when they access the shared data structures. Such overhead can affect not only Active Messages, but traditional two-sided and messages as well. Therefore, it is important to design the communication library carefully in order to minimize the impact of the additional asynchronous thread.

On modern multi-core architecture, shared-memory can be used to improve the communication performance between processes running within the same node. Many communication libraries use shared memory for intra-node communication and use network communication for inter-node communication [68] [69] [62] [70]. Therefore, it is critical for an implementation of Active Messages to use shared-memory to improve performance when possible.

In this chapter, we present a design and implementation of asynchronous processing of MPI-interoperable Active Messages. Our implementation provides asynchronous progress for both one-sided and Active Messages with negligible overhead for two-sided and collective communications. We optimize for the intra-node communication where the sender can directly access the target memory region via shared-memory.

While it is possible to implement Active Messages on top of MPI [66], it is quite difficult to efficiently provide asynchronous progress. The implementation would require a separate thread that can make calls into the MPI library to receive messages and execute the handler. This means that the MPI library would need to run using the MPI_THREAD_MULTIPLE thread level. When an MPI library is running in the MPI_THREAD_MULTIPLE thread level, the library must ensure that every MPI routine is thread-safe which requires the use of mutexes, and thus imposes an overhead on every communication operation. By implementing Active Messages inside the library, it is possible to eliminate this overhead for two-sided and collective communications.

8.1 Classification of Asynchronous Active Messages with MPI Runtime

Active Messages libraries have been previously implemented on top of MPI, such as the design and implementation described in AM++ [66] and AMMPI [67]. Those libraries are built on top of MPI, therefore, they are widely portable, and applications can use MPI functionality simultaneously, like MPI two-sided and collective communications. However, those libraries fail to efficiently support the asynchronous processing of AMs. In order to support asynchronous processing of AMs, those AM library needs to create an additional thread that waits for incoming messages. The MPI library must use the MPI_THREAD_MULTIPLE thread level, which runs in an "active polling" fashion and always occupies CPU even though there is no AMs coming. On the other hand, such design imposes an additional overhead due to the thread synchronization and mutexes used in every communication calls.

We classify the design of asynchronous Active Messages with MPI into three following categories:

- Non-Async: asynchronous processing of AMs is completely not supported.
- Thread-Async: asynchronous processing of AMs is supported by using a thread on top of the MPI library.
- Integrated-Async: asynchronous processing of AMs is provided internally by the MPI implementation.

Both AMMPI and AM++ belongs to the first class: "Non-Async" by default. If the application using AMMPI or AM++ creates a thread to wait for incoming AMs, such usage would fall into the second class: "Thread-Async". In this work, we propose a design that falls into the third class: "Integrated-Async". It can support asynchronous processing of AMs by providing an internal thread and handle AMs in a much more efficient way.

8.2 Design and Implementation of Asynchronous MPI-Interoperable Active Messages

In current MPI implementations, the MPI library invokes a progress engine to process incoming MPI messages, and such progress engine is invoked only when an MPI routine is explicitly called. In order to improve performance for intra-node communication, most MPI implementations use "active polling", in which they do not block while they are waiting for messages. While this strategy can improve the performance of intra-node communication, it has the effect of using the CPU all the time, making CPU busy even when no message is being processed. Some MPI implementations provide asynchronous progress and non-busy-waiting for MPI messages, but these features often come with significant performance penalty.

Traditionally, MPI implementations use a single progress engine to process both one-sided and two-sided messages. Having a single progress engine has two shortcomings: (a) one-sided messages and Active Messages cannot be processed immediately, but have to wait until the target explicitly invokes an MPI routine which internally triggers the progress engine; (b) MPI runtime cannot handle one-sided / Active Messages and traditional two-sided and collective messages in parallel, but has to process them in a sequential manner through the single progress engine.

In our design and implementation, we use two progress engines to process incoming messages: one asynchronous progress engine handling Active Messages and one-sided messages, and one regular progress engine handling traditional two-sided messages and collective messages. Active Messages and one-sided messages can be processed immediately by this separate progress engine and can be processed concurrently with all other kinds of messages. In the following sections, we describe the critical issues in designing and implementing such asynchronous progress engine for both inter-node communication and intra-node communication.

8.2.1 Network Solution

We use an separate thread in the network module to wait for Active Messages and one-sided messages coming from the inter-node communication. When an MPI process meets window creation routine, which indicates that there will be Active Messages or one-sided messages coming, it will internally spawn a separate thread used by the asynchronous progress engine. This thread will be terminated in MPI_FINALIZE. The thread does not wait for messages from intra-node communication, therefore, it can block while waiting for incoming messages with minimal performance overhead. The original progress engine is still used by the main thread to handle two-sided and collective messages.

8.2.2 Shared Memory Solution

Due to the fact that the separate thread invokes the asynchronous progress engine in a blocking manner, it is not a practical design for messages from intra-node communication. We address this by proposing a strategy called "origin computation". When the origin process meets an active message or one-sided message targeting another process on the same node, it directly reads the remote data via shared-memory region, performs the computations (MPIX_AM and MPI_ACCUMULATE) locally and finally pushes results back to the memory on remote process. By using "origin computation", Active Messages and one-sided operations can be handled asynchronously with other messages without the need of a separate thread.

To support the direct access to memory of remote process on the same mode, when window is created at beginning, each MPI process allocates a shared memory region internally and exchanges the shared memory address metadata among all other processes. There are two methods defined in the MPI-3 to create an MPI RMA window and allocate a memory region, which are listed as below. In our framework, we implemented the shared memory solution in both of the following two methods.

- MPI_ALLOC_MEM + MPI_WIN_CREATE: the first routine can allocates a memory region

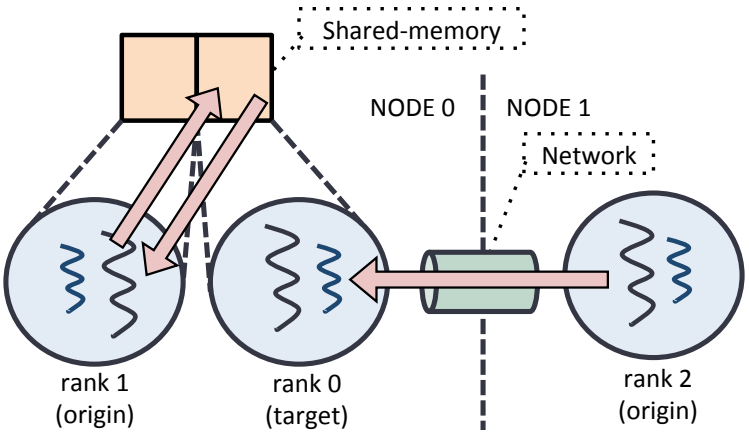


Figure 8.1: Working scenario of asynchronous progress engine

and returns the memory address; the second routine can create an MPI RMA window by using a memory address returned in the first routine.

- MPI_WIN_ALLOCATE: this routine performs memory allocation and window creation together within one single MPI function call.

The application can enable / disable the asynchronous progress engine by using an MPI info argument to the above window creation routines. If the asynchronous progress engine is enabled, the MPI process would internally allocate a shared memory region for the window for intra-node communication and spawn a separate thread for inter-node communication. Figure 8.1 illustrates an example of how the asynchronous progress engine works inside MPI-AM framework. In this example, both rank 0 and rank 2 issues Active Messages to rank 1. Rank 0 is on the same node with rank 1 whereas rank 2 is on a different node. Rank 1 has a separate internal thread to handle Active Messages from rank 2, while rank 0 performs "origin computation" and directly accesses the shared memory region of rank 1. All Active Messages can be performed asynchronously with other MPI messages without busy-waiting on CPU.

8.2.3 Thread Safety and Process Safety

In the network solution, several data structures that are shared by the two different progress engines are either duplicated or protected by mutexes in order to avoid data race condition. In Active Target mode, a counter is employed to decide if all operations have already arrived and if ending RMA synchronization can return. That counter is atomically accessed by two threads. In Passive Target mode, both the main thread and the asynchronous thread on the target process is possible to try to acquire the same lock. We added a mutex on the passive lock on each process to avoid data race condition.

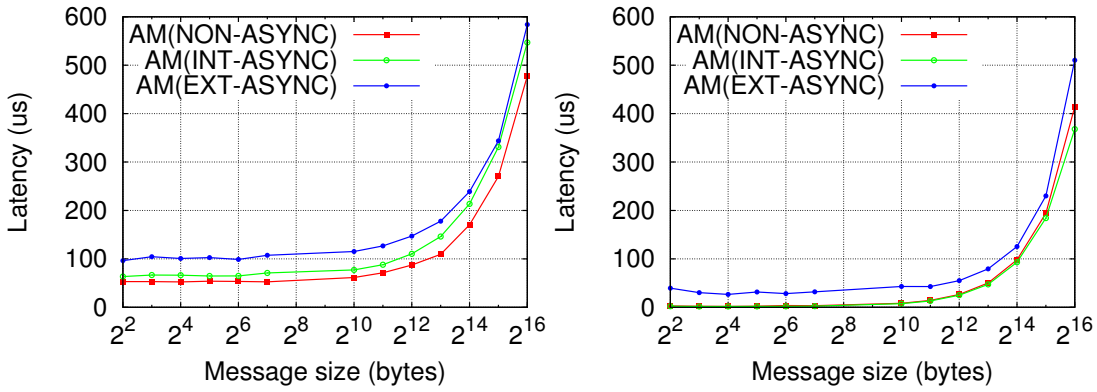
In the shared memory solution, the origin process performs "origin computation" and pushes result data back to the memory on target process within the same node, so it is possible that both the origin process and the target processes modify the same memory location concurrently. We added inter-process mutexes upon the shared memory region on each node to protect such data accesses.

8.3 Experimental Evaluation

In this section, we present the performance results of asynchronous MPI-AM gathered on "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 asynchronous MPI-AM framework based on MPICH2 (1.4).

Here we present the microbenchmark results of latency, overlapping, AM interoperability, and stencil communication. We tested the MPIX_AM, MPI_PUT and MPI_ACCUMULATE, on three different IPC structures: network-only, shared-memory-only, and network-and-shared-memory. All tests are compared under the following three asynchronous execution models:

- EXT-ASYNC: MPI-AM with external asynchronous thread
- INT-ASYNC: MPI-AM with internal asynchronous thread



(a) Inter-node communication (b) Intra-node communication

Figure 8.2: Latency of single AM operation

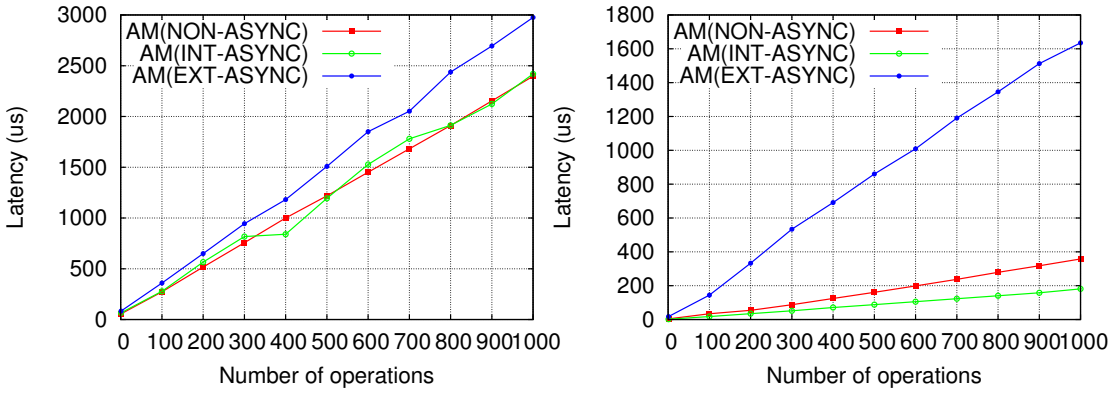
- NON-ASYNC: MPI-AM without asynchronous thread

EXT-ASYNC is enabled by passing an MPI info argument during window creation; INT-ASYNC is enabled by setting the environment variable MPICH_ASYNC_PROGRESS in MPICH, which would spawn a separate thread to actively poll the progress engine.

For tests in Section 8.3.1 and Section 8.3.3, all results are gathered using FENCE synchronization; for tests in Section 8.3.2, all results are gathered using FENCE synchronization for Active Target mode and exclusive LOCK-UNLOCK synchronization for Passive Target mode. Similar results are also observed in other synchronizations.

8.3.1 Communication Latency

Figure 8.2 illustrates the performance comparison of communication latency for a single MPIX_AM operation among the different IPC models when varying message sizes. Figure 8.3 illustrates the latency for multiple MPIX_AM operations among various IPC models when message size is fixed (four bytes). We can see that INT-ASYNC and NON-ASYNC are mostly on par, whereas EXT-ASYNC performance worse. When MPICH_ASYNC_PROGRESS is enabled, MPICH works at MPI_THREAD_MULTIPLE level, making the progress engine become a critical section, and the main thread and the external asynchronous thread must



(a) Inter-node communication (b) Intra-node communication

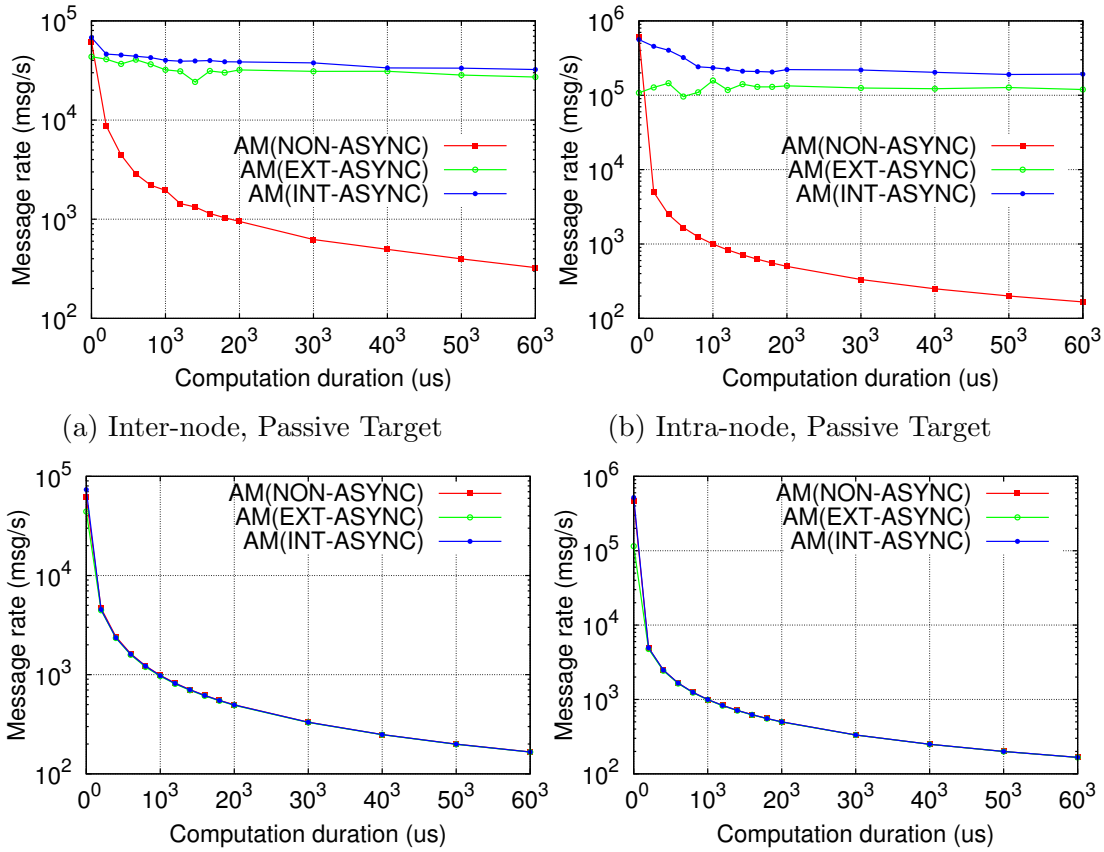
Figure 8.3: Latency of multiple AM operations

compete for it. On the other hand, INT-ASYNC uses a separate progress engine that is not shared with the main thread, therefore there is no critical section involved. The overhead of the EXT-ASYNC thread when entering / exiting the progress engine becomes obvious when the number of operations increases (Figure 8.3). Additionally, we observe that for shared memory communications, INT-ASYNC performs even better than NON-ASYNC case as the number of operations increases (Figure 8.3b), this is because it does direct memory copy instead of copying the messages through the Nemesis [68] send queues in NON-ASYNC.

8.3.2 Overlapping Effects

To measure the performance of overlapping effects of the asynchronous progress engine in MPI-AM, we use tests in which a certain amount of computation is being performed on the target side while the origin side issues a lot of Active Messages. The target process does not invoke any MPI call while it is doing the computation. Figure 8.4 demonstrates the overlapping effects in both Passive Target and Active Target modes.

In Figure 8.4a, both EXT-ASYNC and INT-ASYNC can achieve obvious overlapping effect while NON-ASYNC has no overlapping effect. This is because while the main thread is doing the computation, EXT-ASYNC and INT-ASYNC are able to separately handle communications with the asynchronous progress thread whereas NON-ASYNC can not.



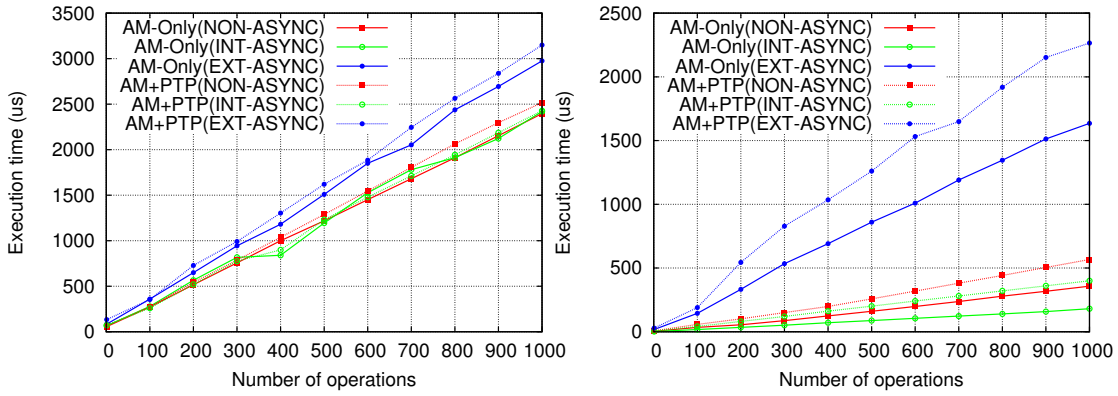
(a) Inter-node, Passive Target (b) Intra-node, Passive Target

(c) Inter-node, Active Target (d) Inter-node, Active Target

Figure 8.4: Overlapping effects of AM asynchronous progress engine

Figure 8.4b illustrates the overlapping effect of the shared memory processing. In EXT-ASYNC model, the asynchronous thread is spawned on the target process to deal with all incoming Active Messages while the main thread is doing local computation; in the INT-ASYNC model, after the origin process acquires the lock on the target side, it can directly perform the computation and writes the result data into the memory of the target while the target is doing local computation. Note that there is a performance gap between EXT-ASYNC and INT-ASYNC on both network and shared memory experiments, because of the queuing operations needed for distributed passive lock.

Results in Figure 8.4a and Figure 8.4b shows that no overlapping effects can be achieved in Active Target mode. Because MPI-AM is implemented based on RMA implementation in MPICH, in which all messages are delayed issued out during the ending synchronization



(a) Inter-node communication (b) Intra-node communication

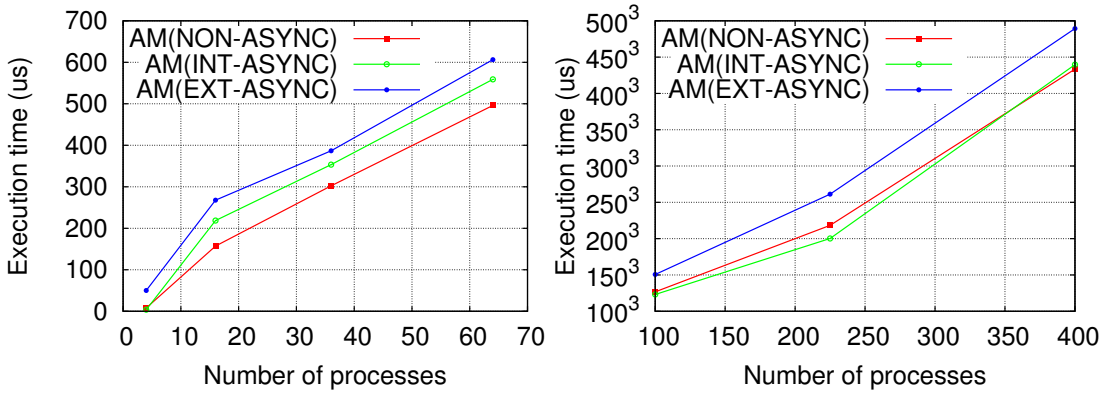
Figure 8.5: Interoperability performance

phase, target receives those messages within the ending synchronization phase, which cannot be overlapped with the local computation within the epoch.

8.3.3 Interoperability Performance

Figure 8.5 shows the execution time when Active Messages and traditional MPI two-sided communications happen together (AM+PTP). In this test, the origin process sends multiple Active Messages to the target process, at the same time, there is a third process sending multiple MPI two-sided messages to the same target process. We increase the number of both AM and two-sided operations and measure the execution time on the origin process.

Figure 8.5a illustrates the interoperability performance on InfiniBand network. When the two-sided communication is added, the execution time of both EXT-ASYNC and NON-ASYNC models rises, because when it is EXT-ASYNC and NON-ASYNC, Active Messages and two-sided messages are processed by the same thread (both are processed by asynchronous thread in EXT-ASYNC and by main thread in NON-ASYNC). However, the time of INT-ASYNC does not increase obviously, because Active Messages and two-sided messages are processed in separate threads in INT-ASYNC: the asynchronous thread processes Active Messages and the main thread processes MPI two-sided messages. Hence, adding two-sided communications would not affect the total execution time in INT-ASYNC.



(a) Small grid size (2×2 to 8×8) (b) Large grid size (10×10 to 20×20)

Figure 8.6: Execution time of stencil kernel benchmark

Figure 8.5b illustrates the interoperability performance on shared memory. When two-sided communication is added, the execution times in all three models are increased, due to the fact that Active Messages and two-sided messages are handled within the same thread for all three cases. However, because there is no extra thread existing in NON-ASYNC and INT-ASYNC, the overhead is relatively small compared to EXT-ASYNC which has an asynchronous thread.

8.3.4 Stencil Kernel Benchmark

In order to evaluate the scalability and the performance of the asynchronous progress on both network and shared memory, we implemented a stencil kernel benchmark using MPIX_AM operations and FENCE synchronization. In this test, each process sends Active Messages to neighbors at distance one. During the execution time, there are four MPI processes on every node. For INT-ASYNC and EXT-ASYNC models, each MPI process spawns a separate thread, therefore all eight cores on the node are being used by different threads.

Figure 8.6a demonstrates the execution time when grid size is relatively small. Both INT-ASYNC and EXT-ASYNC performs worse than NON-ASYNC because of the overhead of additional asynchronous threads, however INT-ASYNC has less overhead than EXT-ASYNC. When grid is 2×2, only shared memory communication exists in the MPI runtime,

so INT-ASYNC performs as fast as NON-ASYNC.

Figure 8.6b illustrates the execution time when grid size is relatively large. From the experimental results, we observe that INT-ASYNC performs nearly the same with NON-ASYNC, and is slightly better for 10×10 and 15×15 grids. The reason is that when more processes are involved in the execution, the synchronization overhead of MPI_WIN_FENCE would increase a lot, which makes the advantage of NON-ASYNC become smaller. When number of processes is increased to 400, INT-ASYNC performs better than EXT-ASYNC as well.

8.4 Conclusion

In this chapter, we describe the design and implementation of asynchronous MPI-interoperable Active Messages based on the MPI RMA interface. The implementation is achieved by using multi-threading and shared memory allocation to support asynchronous progress engine. The impact of this work are as follows: Active Messages can work interoperably with two-sided and collective communications while introducing a modest overhead; a process which does shared memory communication directly performs origin computation which can allow the asynchronous thread to wait for messages from the network without busy-waiting; asynchronous AM is implemented inside MPI, not on top of MPI. Through the evaluation of communication latency, overlapping effect, interoperability and stencil microbenchmarks, we demonstrated that the performance is competitive when comparing with external asynchronous progress engine of Active Messages.

CHAPTER 9

Conclusion

Irregular computation has become increasingly important in recent years, and MPI is the most prominent parallel programming model and runtime for high performance computing. MPI was originally designed for scientific computation where computation is in regular pattern, and it is still unclear if MPI is suitable to newly emerged irregular applications. The goal of this thesis is to evaluate the suitability of MPI to irregular computation.

In this thesis, we conduct the study and research via two subtopics:

- Addressing scalability and performance limitations in massive asynchronous communication
  - Task 1: Tackling scalability challenges in MPI one-sided communication
  - Task 2: Adaptive issuing strategy for MPI one-sided communication
  - Task 3: Scalable virtual connection objects initialization
- Integrated data and computation management
  - Task 4: Generalized MPI-interoperable Active Messages framework
  - Task 5: Optimizing MPI-interoperable Active Messages for different application scenarios
  - Task 6: Asynchronous processing in MPI-interoperable Active Messages

We conclude that MPI runtime and MPI programming model can be suitable to support irregular computation, by proposing efficient solutions to address scalability and performance challenges in massive asynchronous communication, and by proposing new programming model that integrates data and computation management.

The publication related to this thesis is listed below:

- **Runtime Support for Irregular Computation in MPI-Based Applications**. **Xin Zhao**, Pavan Balaji, William Gropp. Doctoral Symposium. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015).
- **Nonblocking Epochs in MPI One-Sided Communication**. Judicael Zounmevo, **Xin Zhao**, Pavan Balaji, William Gropp, Ahmad Afsahi. IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2014). Best Paper Finalist.
- **Optimization Strategies for MPI-Interoperable Active Messages**. **Xin Zhao**, Pavan Balaji, William Gropp, Rajeev Thakur. IEEE International Conference on Scalable Computing and Communications (ScalCom 2013). Best Paper Award.
- **MPI-Interoperable Generalized Active Messages**. **Xin Zhao**, Pavan Balaji, William Gropp, Rajeev Thakur. IEEE International Conference on Parallel and Distributed Systems (ICPADS 2013).
- **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. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013).
- **Adaptive Strategy for One-Sided Communication in MPICH2**. **Xin Zhao**. Master thesis.

- **Adaptive Strategy for One-Sided Communication in MPICH2**. **Xin Zhao**, Gopalakrishnan Santhanaraman, William Gropp. The European MPI Users' Group Meeting (EuroMPI 2012).

- **Scalable Memory Use in MPI: A Case Study with MPICH2**. David Goodell, William Gropp, **Xin Zhao**, Rajeev Thakur. The European MPI Users' Group Meeting (EuroMPI 2011).

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] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. V. Dam, D. Wang, J. Nieplocha, E. Apr, T. L. Windus, and W. A. deJong, "NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations." *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[6] Center for Exascale Simulation of Advanced Reactors, "Cesar," https://cesar.mcs.anl.gov/.

[7] DoE Exascale Co-Design Center for Materials in Extreme Environments, "ExMatEx," http://www.exmatex.org/.

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

[9] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More Scalability, Less Pain: A Simple Programming Model and Its Implementation for Extreme Computing," *SciDAC Review*, vol. 17, pp. 30–37, 03/2010 2010.

[10] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A Framework for Global-View Task Parallelism," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 586–593.

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

[12] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "The Top500 List," http://top500.org/.

[13] 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.

[14] OpenFabrics Alliance (OFA), " OpenFabrics Enterprise Distribution (OFED)," http://www.openfabrics.org/.

[15] The InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1, Release 1.2," 2004.

[16] Sandia National Laboratories, "Portals Network Programming Interface," http://www.cs.sandia.gov/Portals/.

[17] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. Atos, "The BXI Interconnect Architecture," in *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, Aug 2015, pp. 18–25.

[18] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers," *Computer*, vol. 42, no. 11, pp. 36–40, 2009.

[19] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A scalable HPC system based on a Dragonfly network," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–9.

[20] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 26:1–26:10.

[21] InfiniBand Trade Association, "InfiniBand Architecture Specification Release 1.2.1 Annex A17: RoCE v2," https://cw.infinibandta.org/document/dl/7781/, 2014.

[22] Fujitsu Limited, "FUJITSU Supercomputer PRIMEHPC FX100 Evolution to the Next Generation," https://www.fujitsu.com/global/Images/primehpc-fx100-hard-en.pdf.

[23] Argonne National Laboratory, "MPICH," https://www.mpich.org/.

[24] The Open MPI Development Team, "Open MPI," http://www.open-mpi.org/.

[25] Ohio State University, "MVAPICH," http://mvapich.cse.ohio-state.edu/.

[26] R. Thakur, W. Gropp, and B. Toonen, "Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication," *International Journal of High Performance Computing Applications*, vol. 19, pp. 119–128, 2005.

[27] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2013, pp. 53:1–53:12.

[28] D. d. C. Reis, D. Belazzougui, F. C. Botelho, and N. Ziviani, "CMPH Library," https://http://cmph.sourceforge.net/.

[29] F. C. Botelho, D. Belazzougui, and M. Dietzfelbinger, "Compress, Hash and Displace," in *Proceedings of the 17th European Symposium on Algorithms*, ser. ESA 2009. Springer LNCS, 2009.

[30] R. Thakur, W. Gropp, and B. Toonen, "Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication," *INT. J. HIGH PERFORM. COMPUT. APPL*, vol. 19, pp. 119–128, 2005.

[31] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff, "MPI on Millions of Cores," *Parallel Processing Letters (PPL)*, vol. 21, no. 1, pp. 45–60, Mar. 2011.

[32] M. Chaarawi and E. Gabriel, "Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators," in *Proceedings of the 8th International Conference on Computational Science, Part I*, ser. ICCS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 297–306.

[33] H. Kamal, S. M. Mirtaheri, and A. Wagner, "Scalability of Communicators and Groups in MPI," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 264–275.

[34] J. L. Träff, "Compact and Efficient Implementation of the MPI Group Operations," in *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 170–178.

[35] X. Zhao, G. Santhanaraman, and W. Gropp, "Adaptive Strategy for One-sided Communication in MPICH2," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 16–26.

[36] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur, "High Performance MPI-2 One-Sided Communication over InfiniBand," in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, April 2004, pp. 531–538.

[37] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.

[38] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda, "Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 380–387.

[39] J. A. Zounmevo and A. Afsahi, "A Fast and Resource-Conscious MPI Message Queue Mechanism for Large-scale Jobs," *Future Gener. Comput. Syst.*, vol. 30, pp. 265–290, Jan. 2014.

[40] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and Evaluation of Shared-memory Communication and Synchronization Operations in MPICH2 Using the Nemesis Communication Subsystem," *Parallel Comput.*, vol. 33, no. 9, pp. 634–644, Sep. 2007.

[41] W. Gropp and E. L. Lusk, "Reproducible Measurements of MPI Performance Characteristics," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 1999, pp. 11–18.

[42] J. Larsson Traff, H. Ritzdorf, and R. Hempel, "The Implementation of MPI-2 One-sided Communication for the NEC SX-5," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. Washington, DC, USA: IEEE Computer Society, 2000.

[43] Cray Research Inc, "Cray T3E C and C++ optimization guide," 1994.

[44] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems," *Proceedings of the Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP*, Apr. 1999.

[45] GASNet, "GASNet," http://http://gasnet.cs.berkeley.edu/.

[46] M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas, "Portable and Efficient Parallel Computing Using the BSP Model," *IEEE Trans. Comput.*, vol. 48, no. 7, pp. 670–689, July 1999.

[47] J. M. Hill and D. B. Skillicorn, "Lessons Learned from Implementing BSP," *Future Gener. Comput. Syst.*, vol. 13, no. 4-5, pp. 327–335, Mar. 1998.

[48] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "BSPlib: The BSP Programming Library," *Parallel Comput.*, vol. 24, no. 14, pp. 1947–1980, Dec. 1998.

[49] W. D. Gropp and R. Thakur, "Revealing the Performance of MPI RMA Implementations," 2007, pp. 272–280.

[50] W. Gropp and R. Thakur, "An Evaluation of Implementation Options for MPI One-Sided Communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. D. Martino, D. Kranzluüller, and J. Dongarra, Eds., no. LNCS 3666. Springer Verlag, Sep. 2005, 12th European PVM/MPI User's Group Meeting, Sorrento, Italy. pp. 415–424.

[51] R. Thakur, W. Gropp, and B. Toonen, "Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., no. LNCS3241. Springer Verlag, 2004, 11th European PVM/MPI User's Group Meeting, Budapest, Hungary. pp. 57–67.

[52] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, "High Performance MPI-2 One-Sided Communication over InfiniBand," in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, April 2004, pp. 531–538.

[53] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. D. Gropp, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19 - 22, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters, pp. 68–76.

[54] G. Santhanaraman, S. Narravula, and D. K. Panda, "Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–11.

[55] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda, "Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 380–387.

[56] J. L. Traff, W. D. Gropp, and R. Thakur, "Self-Consistent MPI Performance Guidelines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 698–709, May 2010.

[57] X. Zhao, "Adaptive Strategy for One-Sided Communication in MPICH2," M.S. thesis, University of Illinois at Urbana-Champaign, 7 2012.

[58] J. Barbay and G. Navarro, "Compressed Representations of Permutations, and Applications," in *26th International Symposium on Theoretical Aspects of Computer Science*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Albers and J.-Y. Marion, Eds., vol. 3. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 111–122.

[59] A. Woo, D. Bailey, M. Yarrow, W. Wijngaart, T. Harris, and W. Saphir, "The NAS Parallel Benchmarks 2.0," The Pennsylvania State University CiteSeer Archives, Tech. Rep., Dec. 1995.

[60] "ASC Sequoia Benchmark Codes: AMG," 2011, https://asc.llnl.gov/sequoia/benchmarks/#amg.

[61] D. Goodell, W. Gropp, X. Zhao, and R. Thakur, "Scalable Memory Use in MPI: A Case Study with MPICH2," in *EuroMPI*, 2011, pp. 140–149.

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

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

[64] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP," in *International Parallel Processing Symposium*, 1998.

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

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

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

[68] D. Buntinas, G. Mercier, and W. Gropp, "Design and Evaluation of Nemesis, a Scalable Low-Latency Message-Passing Communication Subsystem," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Washington, DC, USA, May 2006, pp. 521–530.

[69] 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.

[70] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems," *Workshop on Runtime Systems for Parallel Programming (RTSPP); International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.