NODE-WIDE ASYNCHRONOUS MESSAGE PROGRESSION FOR EFFICIENT & SCALABLE COMMUNICATION IN HIGH PERFORMANCE COMPUTING

By

KAUSHAL KUMAR

A thesis submitted to the

Department of Electrical & Computer Engineering

in conformity with the requirements for

the degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

(December, 2016)

Abstract

High Performance Computing (HPC) has served as the enabler for several scientific and engineering accomplishments. Consequently, there has been an ever-increasing demand to create larger and faster high performance systems. To efficiently utilize the HPC resources, parallel applications rely on software that abstract the cluster hardware. Currently, the most prominent software abstraction standard in HPC is the Message Passing Interface (MPI). Parallel applications entail communications to synchronize and to share intermediate results. To minimize the duration of application executions, it is crucial to overlap such communications with the computations.

The MPI standard specifies three messaging semantics, namely, point-to-point communication, one-sided communication and collectives. In point-to-point communication, the overlap of large payloads has been a problem and there have been several proposals to address this. Among these approaches, asynchronous message progression is more adoptable because of its ability to deal with a wider range of inefficiencies and its non-reliance on specialized hardware. Traditional asynchronous message progression approaches have relied on either polling or interrupt based threads. The polling based approach is more responsive but is resource-intensive. On the other hand, the interrupt based approach is resource-efficient but is associated with overheads. This thesis proposes a node-wide asynchronous message progression technique that offers the advantages of both polling and interrupt based approaches, while minimizing or eliminating their adverse effects. This approach was found to be scalable, incur negligible overheads, induce the ideal amount of overlap in most scenarios of point-

to-point communications and cast a small memory footprint. This technique was found to improve the overlap of certain collectives as well.

One-sided MPI communication offers the ability to transfer messages with few or no synchronizations, regardless of the payload size. This scheme promotes overlaps but there are several overlap inhibiting scenarios. This thesis proposes a similar asynchronous message progression technique to address such scenarios. The one-sided implementation was able to achieve overlap in the different inefficient scenarios, with negligible overheads and a small addition to the memory footprint.

Acknowledgements

First and foremost, I would like to extend my sincerest gratitude to my supervisor, Dr. Ahmad Afsahi, without whom this dissertation would not have been possible. I am deeply indebted to him for his constant guidance, encouragement and unwavering support over the last two years. I am also indebted to Dr. Judicael Zounmevo for his insights and for the knowledge that he has imparted to me. My heartfelt gratitude to the members of the thesis defense committee, Dr. Thomas Dean, Dr. Ying Zou and Dr. Steven Blostein for their insightful comments and questions. I would like to thank the Natural Science and Engineering Research Council of Canada (NSERC) for supporting this research through grants to my supervisor. I am also thankful to Queen's University for supporting me financially through awards and teaching assistantships.

I would like to thank the HPC Advisory Council and Mr. Pak Lui for providing the resources and technical support for this research. I am also very grateful to our Graduate Program Assistant, Ms. Debra Fraser for helping me with academic and administrative matters.

I would like to express my heartfelt gratitude to my colleagues at the Parallel Processing Research Laboratory for the wonderful times that we have shared. Iman, Hessam, Mahdieh, Mac and Priya, I will cherish your friendships forever. I also thank my fellow graduate students for their company and the thought-provoking discussions.

Last but not the least, I would like to thank my parents for their patience and unquestioning support over the years.

Table of Contents

Abstract ii

Acknowledgements iv

Table of Contents v

List of Figures ix

Glossary xii

Chapter 1: Introduction 1

1.1 Motivation 3

1.2 Problem Statement 5

1.3 Contributions 7

1.4 Outline 9

Chapter 2: Background 11

2.1 HPC Clusters 12

2.1.1 Remote Direct Memory Access and InfiniBand 13

2.1.2 NUMA Based Compute Node 16

2.1.3 Parallel Programming Paradigms 17

2.2 Shared Memory Model 18

2.3 Partitioned Global Address Space Model 19

2.4 Message Passing Model 19

2.4.1 Point-to-point Communication 20

2.4.2 Collective Communications 27

2.4.3 One-Sided Communication 29

2.5 Summary 37

Chapter 3: Literature Review 39

3.1 Point-to-Point Communication 40

3.1.1 Inefficiencies Associated with the Rendezvous Protocols 40

3.1.2 Literature Review 44

3.2 One-Sided Communication or Remote Memory Access (RMA) 53

3.2.1 Inefficiencies Associated with RMA Synchronizations 53

3.2.2 Literature Review 56

3.3 Summary 57

Chapter 4: Node-Wide Asynchronous Message Progression for Point-to-Point Communication 60

4.1 Motivation 61

4.2 Design of SmartInterrupts 62

4.2.1 Choosing the Default Rendezvous Protocol 66

4.2.2 Asynchronous Message Progression Mechanism 66

4.2.3 Core Components 68

4.3 Implementation of SmartInterrupts 69

4.3.1 Interrupt Handler Kernel Module 69

4.3.2 Shared Buffers 70

4.3.3 Helper Process and Interrupt Thread 73

4.4 Design Alternatives 76

4.5 Performance Evaluation and Analysis 78

4.5.1 Description of Hardware and Software 78

4.5.2 Two-Sided Micro-Benchmarks 78
4.5.3 Collective Micro-Benchmarks 87
4.5.4 Application Results 92
4.6 Summary 94
Chapter 5: Node-Wide Asynchronous Message Progression for One-Sided Communication 96
5.1 Motivation 97
5.2 Design of SmartInterrupts for One-Sided Communications 101
5.2.1 Asynchronous Message Progression Mechanism 103
5.2.2 Core Components 106
5.3 Implementation of One-Sided SmartInterrupts 107
5.3.1 Shared Buffers 107
5.3.2 Helper Process and Interrupt Thread 108
5.4 Performance Evaluation and Analysis 109
5.4.1 Fence and GATS One-Sided Micro-Benchmarks 110
5.4.2 Exclusive Lock One-Sided Micro-Benchmark 116
5.4.3 Memory Footprint Analysis 121
5.4.4 Application Results 122
5.5 Summary 123
Chapter 6: Conclusion and Future Work 126
6.1 Summary of Findings 127
6.2 Future Work 129
6.2.1 Unifying One-Sided and Two-Sided Designs 129
6.2.2 Improving the Performance of SmartInterrupts for Collectives 130
6.2.3 Eliminating Futile Interrupts in One-Sided SmartInterrupts 131

6.2.4 Dynamically Enabling/Disabling the SmartInterrupts Mechanism 131

6.2.5 Investigating Multi-threaded MPI Applications 132

Bibliography 133

List of Figures

2.1 Illustration of a Commodity Cluster 13

2.2 Comparison of Copy Mechanisms between TCP/IP and RDMA 14

2.3 Anatomy of a NUMA Compute Node 17

2.4 Sender Initiated Rendezvous Protocols (Sender Arriving First Scenario) 22

2.5 Hierarchy of Buffers in Common MPI Implementations 24

2.6 Comparison of Communication/Computation Overlap between Blocking and Non-Blocking Two-Sided Calls in RDMA Read Rendezvous Protocol 28

2.7 MPI 3.1 RMA Synchronizations 31

2.8 Comparison of Unproductive Waits with Blocking and Non-Blocking RMA Synchronization Calls 36

3.1 Sender Initiated RDMA Read Based Rendezvous Protocols 41

3.2 Sender Initiated RDMA Write Based Rendezvous Protocols 42

3.3 Receiver Initiated RDMA Write Based Rendezvous Protocols 45

3.4 Interrupt Thread based Asynchronous Progression 52

3.5 Illustration of Parasitic Execution Flow Based Asynchronous Message Progression (Adapted from [87]) 53

4.1 Data Movement and Signals in Smart Interrupts 67

4.2 Core Components of SmartInterrupts 69

4.3 Illustration of Buffers and Data Movement in SmartInterrupts 71

4.4 Pair-wise Communication in Two-Sided Micro-Benchmarks 79
4.5 Template for Two-Sided Micro-Benchmark Design 80
4.6 Two-Sided Latency Overhead Results over MVAPICH 82
4.7 Communication/Computation Overlap of Point-to-Point Communications 84
4.8 Scalability Analysis of Two-Sided SmartInterrupts 85
4.9 Asynchronous Message Progression for 9 MPI processes per node, 1 Helper Process for Smart Interrupts 87
4.10 Asynchronous message progression for 18 MPI processes per node, 2HPs for SmartInterrupts 88
4.11 Template for Collective Micro-Benchmark Design 89
4.12 MPI_Ialltoall Overlap and Asynchronous Message Progression Results 90
4.13 MPI_Igather Micro-benchmark Results 92
4.14 NAS-SP Results 93
5.1 Inefficiencies with Non-blocking RMA Synchronizations 98
5.2 Illustration of Communication/Computaiton Overlap with Shared Lock Epochs 100
5.3 Data Movement and Signals in One-Sided SmartInterrupts 104
5.4 Communication Patterns Used in One-Sided Micro-Benchmarks 111
5.5 Template for One-Sided Latency Overhead Micro-Benchmark 113
5.6 One-Sided Latency Overhead Results with Fence and GATS 114
5.7 Template for Fence & GATS Communication\Computation Overlap Micro-Benchmark 115
5.8 One-Sided Overlap Results for Fence Epochs with Pair-Wise Communication Pattern and 1 HP 117
5.9 One-Sided Overlap Results for GATS Epochs with Pair-Wise Communication Pattern and 1 HP 118

5.10 One-Sided Overlap Results for Fence Epochs with One-to-Many Communication Pattern and 1 HP 119
5.11 One-Sided Overlap Results for GATS Epochs with One-to-Many Communication Pattern and 1 HP 120
5.12 Template for Exclusive Lock Communication\Computation Overlap Micro-Benchmark 121
5.13 Overlap Micro-benchmark Results for Exclusive Lock Epochs with 1 HP 121
5.14 LU Decomposition Results with 1 Helper Process per Node 123

Glossary

API Application Programming Interface 2

CQ Completion Queue 15

CTS Clear To Send 22

FLOPS Floating-Point Operations Per Second 1

GATS General Active Target Synchronization 30

GPU Graphics Processing Unit 2

HCA Host Channel Adapter 15

HP Helper Process 65

HPC High Performance Computing 1

HWLOC Portable Hardware Locality 75

ICB Interrupt Control Buffer 70

ICD Interrupt Control Data 71

IPC Inter-Process Communication 16

IRB Interrupt Request Buffer 71

IRD Interrupt Request Data 73

MP MPI Process 78

MPP Massively Parallel Processing 2

MPI Message Passing Interface 3

NAS NASA Advanced Supercomputing 92

NIC Network Interface Card 3

OFED OpenFabrics Enterprise Distribution 70

OS Operating System 13

PCIe Peripheral Component Interconnect Express 13

PGAS Partitioned Global Address Space 19

PMPI MPI Profiling Interface 52

POSIX Portable Operating System Interface 19

PRQ Posted Receive Queue 23

QP Queue Pair 15

RDMA Remote Direct Memory Access 11

RMA Remote Memory Access 4

RTR Request To Receive 44

RTS Ready To Send 21

SGE Scatter Gather Element 15

SP Scalar Pentadiagonal 92

SPE Synergistic Processing Elements 46

SRQ Shared Receive Queue 48

TCP/IP Transmission Control Protocol/Internet Protocol 13

UMQ Unexpected Message Queue 23

UPC Unified Parallel C 19

VBUF Virtual Buffer 70

VmRSS Virtual Memory Resident Set Size 86

WR Work Request 15

Chapter 1

Introduction

The speed of the computations can be enhanced by increasing the clock frequency of the processor, however, only up to a certain limit. However, power and heat dissipation requirements make it impractical to fabricate very high frequency processors [53]. Also, certain applications in science, engineering and business domains execute complex algorithms and work on large datasets that may take years, decades or even centuries to run on a single processor. The solution to this is parallel computing, in which the main problem is divided into smaller subsets and the sub-problems are solved on different processors. High Performance Computing (HPC) uses this idea to solve computationally-intensive problems on systems consisting of hundreds, thousands or even millions of CPU cores [7]. It may not be apparent, but HPC is ubiquitous. Today, HPC is a vital component of several scientific, industrial and commercial fields such as physics (nuclear, astrophysics, applied, particle, aerodynamics), biochemistry (cancer and drug research), geology, mathematics, defense, medical imaging and diagnosis, financial trading and climate modelling, to name a few.

The performance of a computer is often measured in terms of **floating-point operations per second (FLOPS)**. **Top500** [80] is an organization that uses this yardstick to rank the fastest supercomputers of the world. At the time of writing, this list featured 95 supercomputers that are capable of delivering petascale performance. Meaning, that they can perform over quadrillion (10^{15})

floating-point operations per second. The fastest supercomputer in the November, 2016 rankings was Sunway TaihuLight with over 10 million cores and about 93 Petaflops of compute power. Because of the ever increasing demand for HPC, this list is not expected to stagnate anytime soon, in fact, the next milestone is to reach **exascale** [3] by 2023. In the past, the high procurement and maintenance costs limited HPC access to large government agencies and wealthy corporations. The scenario today, however, is quite different due to the widespread adoption of commodity computing. This is a cost effective solution in which an HPC system can be assembled from off-the-shelf computer components instead of choosing proprietary options. Such a system is called a cluster, which is comprised of several compute nodes connected to each other by means of a high-performance network. Among other components, each compute node contains one or more processors and can be supplemented with Graphics Processing Units (GPUs) and many-core co-processors such as Intel Xeon Phi [69]. Clusters are becoming increasingly prevalent in HPC, in fact, at the time of writing, 86 percent of the Top500 supercomputers were found to be clusters. The rest were Massively Parallel Processing (MPP) systems which are proprietary solutions.

In a cluster, the computations may be performed locally on each compute node but they communicate with each other to synchronize and exchange intermediate results. This exchange of information is accomplished by complex networking code, on top of high-speed networks, that use low-level Application Programming Interfaces (APIs). To alleviate the application programmers from this daunting task, parallel programming paradigms (or models) [6] have been introduced. Standardized programming models are also useful to facilitate the portability of application code as different clusters may have different underlying hardware. Not surprisingly, researchers think that the race to exascale computing requires as much attention on the parallel programming paradigms as the underlying hardware. One of the most prominent paradigms in HPC, that has been around for decades is the *message passing* paradigm, in which the data is exchanged between individual processes by explicit communication and synchronization. The **Message Passing**

Interface (MPI) standard is the flag-bearer of the message passing paradigm and is governed by the MPI Forum [54]. It specifies different messaging semantics and a set of API calls to facilitate communications between processes. Since its inception in 1992, the MPI standard has seen widespread acclamation by the HPC community, and it is expected to continue with its success in the years to come [29]. The flexibility that it offers and its suitability for HPC systems make it unlikely to be replaced by other paradigms in the near future. MPI can also be combined with other programming models such as shared memory [64] and PGAS [21] to form hybrid paradigms which are referred to as MPI + X, X referring to other programming models.

1.1 Motivation

When a serial application is parallelized, its speedup may not be equal to the amount of increase in the computational power. In other words, increasing the number of processors from one to one hundred cannot guarantee the execution time to come down to a hundredth. This is because of the necessary communications between the processors to exchange data and arrive at the final result. Therefore, the communication latency can significantly influence an application's execution time. Also, MPI applications often contain chains of processes that depend on each other. In such a situation, a delay at one process can easily propagate to the rest of the processes in the chain. Such an effect can be devastating to the performance, especially when large clusters are involved.

The frequency and volume of communications is largely dictated by the application and reducing such parameters is beyond the scope of the MPI standard. However, an effective usage of the MPI calls can improve the application performance by hiding the communication latency. That is, the MPI calls can be issued in a way such that the communication progresses in parallel with the computation. This *communication/computation overlap* is facilitated by modern interconnects, such as Intel Omni-Path [11], InfiniBand [36], iWARP Ethernet and BXI [20], in which the communication primitives can be offloaded to the **network interface card (NIC)** and the CPU

does not have to be involved throughout the length of the communication. Effective usage of MPI calls can provide the means to improve the overlap but there are scenarios where the serialization of communication and computation cannot be avoided by the application programmer. In some of these scenarios, the fundamental messaging semantics of MPI are not enough to hide the latencies of the communications. Such scenarios require optimizations to the MPI middleware or require augmenting it with support mechanisms to induce communication/computation overlap which would have not been possible with a reference MPI implementation.

The MPI standard specifies three types of communication semantics, namely, two-sided or point-to-point communications, one-sided or RMA communications and collectives. The details of these communication semantics are discussed in Chapter 2. Two-sided communication is based on a send-receive model, in which both the sender and the receiver have to explicitly issue MPI calls for a successful message transfer. On the other hand, one-sided communication is based on the direct access to the remote peer's buffer. Transfer of one-sided messages do not require MPI calls to be issued by all the involved peers. In fact, only the peer that intends to read or write data to the other peer is required to issue the MPI call. However, the transfer of an RMA message cannot be initiated until the peers have synchronized amongst themselves. This synchronization essentially grants permissions for remote memory operations. A synchronization is required again after all the RMA communications are complete. Finally, the MPI standard specifies API calls for some special communications patterns that are frequently used in parallel applications. These functions are called collectives and are usually used when more than two peers are involved. Collective calls can often be replaced by a group of two-sided or one-sided calls, however, the use of the former is generally suggested to take advantage of the middleware optimizations.

There are a large number of scientific applications based on MPI, and a huge percentage of these applications have a large codebase. Such applications are often executed over extended periods of time, therefore, the occurrence of communication/computation serializations is not

implausible. Also, because of the nature of such applications and the scale at which they are executed, even small improvements in communication/computation overlap can have a huge overall impact on the performance.

1.2 Problem Statement

In the context of two-sided MPI communication, a blocking call is a function that does not return until the communication is complete. Use of a blocking call ensures the serialization of computation and communication, but it cannot be avoided in certain scenarios. Also, with modern high-speed interconnects, the latency of small messages (some bytes) does not have a huge impact on the performance if it gets serialized [22]. Therefore, using blocking calls for small messages is acceptable. But, in order to observe any overlap in the application, the use of non-blocking calls is mandatory. A non-blocking two-sided call tries to initiate the communication immediately if possible, otherwise, it defers the communication for later and returns. This message is then expected to transfer when the process is involved in other activities. However, each non-blocking call has to be ultimately blocked at some point. If the message transfer happens before that time, then there is negligible waiting at that blocking call.

Point-to-point messages can be exchanged using two protocols, namely, *eager* and *rendezvous*. In the eager protocol, the sender sends the message directly to the receiver, without the need for any synchronization. On the other hand, the rendezvous protocols require synchronizations by means of control signals. With the eager protocol, there is always a 100 percent overlap but the assumption is that there will always be enough amount of registered buffer available at the receiver [34], which is an unrealistic assumption for large messages (above a few kilobytes). So, large messages are often sent using the rendezvous protocol, in which a co-ordination between the peers is required before the transfer of the message. Overlap can be achieved at either or both of the peers if certain conditions are met [67], such as the timely issuance of MPI calls and presence of enough

computation to overlap the communication with. In a reference MPI implementation, however, there might be no overlap even if these conditions are met [76]. This is possible when the control signal arrives while the process is involved in a computation. While it is busy, the control signal remains unacknowledged by it. So, even though the message transfer conditions were met, the transfer could not be initiated immediately, leading to a potentially impaired overlap.

As mentioned previously, the transfer of RMA messages cannot be initiated until there is a synchronization between the peers. A synchronization is also required after the completion of all the RMA operations. The calls for the transfer of RMA messages is always non-blocking, however, the synchronization calls can be blocking or non-blocking. Blocking synchronizations have been shown to impede communication/computation overlaps [33, 45, 88], therefore, the use of non-blocking RMA synchronizations is vital to observe overlaps [88]. Like two-sided communication, certain conditions need to be satisfied in order to overlap RMA communications with computations. If a peer does not synchronize timely or if it does so when the other peer is busy, then the RMA communications may end up being serialized. Therefore, the use of non-blocking calls does not guarantee overlaps in all scenarios. Also, unlike two-sided communications, there is no upper limit on the number of MPI processes that may be involved in RMA synchronizations, so the unproductive delays may get compounded and propagated to several other peers. This may have a significant impact on the overall performance of the application.

This thesis focuses on the above mentioned inefficiencies and tries to answer the following research questions:

1. What are the exact scenarios that inhibit communication/computation overlap in rendezvous protocols for point-to-point communications and RMA synchronizations for one-sided communication? What is the state-of-the-art in supporting communication/ computation overlap in rendezvous protocols as well as RMA synchronizations?

2. Is it possible to develop a node-wide message progression technique that could achieve the highest possible communication/computation overlap in all scenarios for point-to-point communications? Is it possible for such a solution to be deterministic, scalable, resource efficient and with negligible overhead? Can parallel applications benefit from such a solution?
3. Is it possible to extend such a communication/computation overlap technique to one-sided RMA communications? Does such a solution provide a low-overhead, scalable, and resource-efficient overlap approach for one-sided communications and applications that use them?

1.3 Contributions

From the last section, it can be inferred that both two-sided and one-sided communications suffer from similar inefficiencies. Not surprisingly, there are similar categories of solutions for these inefficiencies, such as, protocol improvement approaches [51, 61, 66, 74] for two-sided communication or non-blocking RMA synchronizations [88], hardware-assisted approaches [20, 27, 70] and host-based approaches [34, 73, 76, 87]. One common inefficiency occurs when the send/receive or the RMA synchronization function is not called timely, that is, the other peer is so late that there is not enough computation to overlap with. The timely arrival of the peers depends a lot on the application's algorithm. Altering this at the middleware could alter the behaviour of the application itself, therefore, the application programmer must take care to avoid such situations as much as possible. Certain solutions exist for two-sided communications [51, 74] but they are associated with issues of their own. The details of these approaches are discussed in Chapter 3.

The other scenario in which an overlap might be prevented is when one of the peers tries to synchronize while the other peer is busy in a computation. In this case, the transfer of communications cannot initiate before the end of the latter's computation. One solution that has

been proven to work well for this scenario is asynchronous message progression [34, 73]. The idea behind this approach is to have a thread that is dedicated for progressing the communications. If a peer issues the send/receive or RMA synchronization call while the application threads of the other peer are busy, then the communications are initiated immediately by the asynchronous progression thread of the latter.

Chapter 3 discusses the asynchronous progression techniques that have been proposed in the literature. The problem with the current techniques is that they are either resource intensive or associated with several overheads. Also, there is no practical solution that works well for both two-sided and one-sided communications. Bearing the limitations of the current techniques in mind, this dissertation makes the following contributions:

1. The idea presented in this thesis is a novel asynchronous message progression solution that can improve the overlap of two-sided communications, one-sided communications, as well as some collectives.
2. The existing approaches for point-to-point communication are either based on interrupts [46, 76] or polling [34]. Interrupt based approaches are resource efficient but are associated with several overheads. Polling based approaches are more responsive but since the existing techniques propose the usage of one progression thread per MPI process, this can either lead to the wastage of compute resources or cause oversubscription. The approach proposed in Chapter 4 is a node-wide message progression technique which utilizes the strengths of both polling and interrupt based approaches, while avoiding their shortcomings. It uses the available spare resources to launch processes that aid in message progression. This makes it suitable for modern compute nodes that are equipped with many-core and multi-core devices. Also, this is a deterministic approach, so it is immune to the overheads that are caused due to the erratic arrival order of the peers.

3. The lack of two-sided node-wide progression techniques meant that the existing micro-benchmarks could not be used to evaluate the new proposal. Chapter 4 describes a set of new micro-benchmarks for two-sided communications where multiple processes of a node may communicate at the same time.
4. The existing overlap techniques for one-sided communication address only a subset of the inefficiencies. Also, most these approaches work exclusively for one-sided communication only. The approach proposed in Chapter 5 is a node-wide asynchronous message progression technique that is implemented on top of NewRMA [86, 88]. NewRMA has been proven to address several RMA inefficiencies, therefore, the proposed implementation improves NewRMA and address a large subset of inefficiencies.

1.4 Outline

The rest of the thesis is divided into 5 chapters. Chapter 2 provides the background and lays the foundation for the following chapters. It starts by briefly discussing the different parallel programming paradigms like message passing, shared memory and PGAS. It then focuses on MPI and describes the different message passing semantics specified in the MPI standard. Chapter 3 discusses the inefficiencies that prevent communication/computation overlaps in MPI and presents a literature review of the existing approaches. Specifically, it presents an in-depth analysis of the rendezvous protocols and RMA synchronizations, and highlights the scenarios where the overlap is inhibited. Chapter 4 proposes a novel approach for improving the overlap of two-sided MPI communications. It is an asynchronous message progression technique called SmartInterrupts. This chapter describes the design and implementation of SmartInterrupts, followed by its performance evaluation. Chapter 5 extends the design of SmartInterrupts to one-sided communications. It highlights the issues that are yet to be efficiently handled in the current approaches and addresses them through SmartInterrupts. It then reports the performance results of the micro-benchmarks and

an application that were used to evaluate the proposed design. Finally, this thesis is concluded in Chapter 6 which remarks on the future scope of the research presented in this document.

Chapter 2

Background

Scientific and engineering applications execute complex algorithms that would take an impractical amount of time to run on a single processor. The executions of such applications may be sped up by using HPC, which utilizes parallel computing to run these applications on systems comprising of hundreds, thousands or millions of processors. The compute power of HPC systems are generally measured in terms of floating-point operations per second or FLOPS. This metric is used by Top500 [80] to rank the fastest supercomputers of the world. Currently, there are several petascale (10^{15}) systems on this list and some of them employ millions of processors. However, the demand for HPC is immense and its horizons are ever-expanding. In fact, there is a strong drive among the HPC community to achieve exascale computing [3] by 2023.

This chapter is intended to provide the relevant background for the rest of the discussion in this thesis. It will introduce the HPC hardware, with a detailed discussion on RDMA and InfiniBand. This will be followed by introducing the different parallel programming paradigms such as shared memory, PGAS and message passing. The focus will then be concentrated on MPI and its different messaging semantics such as point-to-point communication, collectives and one-sided communication. With respect to these messaging semantics, this chapter also aims to discuss important concepts such as message matching, message progression and communication/ computation overlap.

2.1 HPC Clusters

In the past, access to HPC systems was limited to government agencies and big corporations. One of the reasons for this is the relatively high monetary costs associated with proprietary solutions offered by companies such as IBM and Cray. The scenario today, however, is quite different due to the advent of commodity cluster computing. Clusters are associated with lower procurement and maintenance costs, and can be assembled using widely available components. Currently, 86 percent of the Top500 supercomputers are clusters. As illustrated in Figure 2.1, a cluster consists of several compute nodes that are connected to each other through an *interconnect*. The computations are performed on the compute nodes which use the interconnect to share the intermediate results or to synchronize. In the figure, the components inside the compute nodes that are highlighted in blue are mandatory and the ones in green are optional. The CPUs are the primary compute elements and may be supported by GPUs and co-processors. GPUs and co-processors are becoming increasingly prevalent in modern clusters. Compared to the CPUs, these have a large number of relatively low power cores and are ideal for some massively parallel applications.

As mentioned in Chapter 1, the speedup of a serial application may not scale linearly when it is parallelized. Also, increasing the parallelization usually has a diminishing return on the speedup. This is because of the necessary communications and synchronizations between the parallel tasks. In case of a cluster, this entails inter-node communications through the interconnect. In order to achieve maximum speedup, these interconnects not only have to be fast but also provide special features to optimize the communications. One such feature is to support the offloading of the communications to the network interface card (NIC). This reduces the involvement of the CPU in communications and allows it to spend more time on computations; thereby providing the possibility of overlapping the communication latencies with the computations. Today, the most prevalent high performance interconnect is **InfiniBand** [36]. In fact, at the time of writing, more

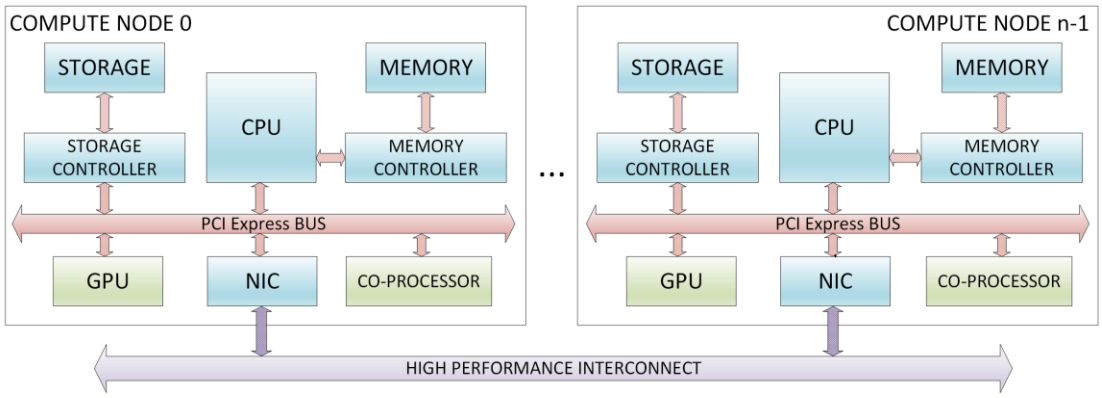


Fig. 2.1. Illustration of a HPC Cluster

than 40 percent of the top 500 fastest supercomputers use InfiniBand [80]. Other high performance interconnects include Intel Omni-Path Fabric [11], Bull eXascale Interconnect (BXI) [20], iWARP Ethernet and RoCE for clusters, and Cray Aries [24] and IBM PERCS [5] for MPP Systems. Because of its popularity, this thesis frequently refers to InfiniBand; however, the discussions and proposals presented herein are not bound to it and can be extended to other interconnects. The primary reason behind this is **Remote Direct Memory Access (RDMA)** [68], which is the common enabling technology behind most of the high performance interconnects.

2.1.1 Remote Direct Memory Access and InfiniBand

Over the years, RDMA has become synonymous with HPC networking. RDMA capable networks enable low-latency and high-throughput communications, which are desirable traits for the current petascale and future exascale systems. RDMA supports zero-copy semantics through which data can be moved between the communicating peers, without the involvement of the operating system (OS). Figure 2.2 compares the zero-copy mechanism of RDMA with the copy mechanism involved in a more traditional protocol such as TCP/IP (Transfer Control Protocol/Internet Protocol). As can be seen in the figure, TCP/IP requires copying the data at

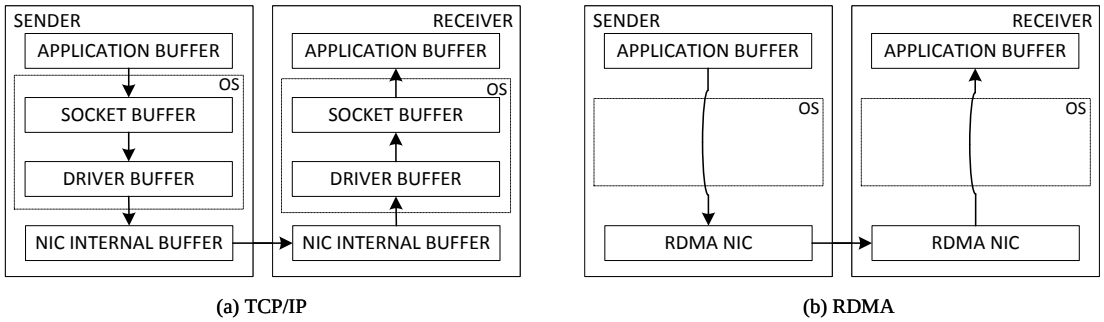


Figure 2.2. Comparison of Copy Mechanisms between TCP/IP and RDMA

multiple intermediate locations at both the sender and the receiver, before the data reaches the target application. All these intermediate transfers require the participation of the CPU. This is not suitable for HPC applications as it would cause frequent disruptions in the computations. RDMA, however, directly deals with the address of the remote application buffer, and the data transfer is carried out by the NICs themselves. The CPU is only involved for a very short time interval to add the message transfer request to the NIC. Also, in case of a traditional interconnect such as Ethernet, the operating system is the sole owner of the NIC. Therefore, the sending application cannot access the NIC directly and must depend on the operating system to relay the information to the NIC. On the other hand, the receiving application must depend on its operating system to relay the messages that arrive on its NIC and are addressed to it. RDMA offers OS-bypass mechanisms through which the applications can directly access the NIC and send or receive data without involving the operating systems at either ends.

One of the most popular networking standards that supports RDMA natively is InfiniBand [36]. The high performance and versatility delivered by InfiniBand makes it the interconnect of choice in HPC. InfiniBand provides two transfer semantics; a channel semantic called **Send/Receive** and a memory semantic supporting **RDMA Read** and **RDMA Write** operations. In the channel semantic, the sending side does not have any information about the receiver's buffer address. It requires a data-structure to be pre-posted at the receiver, so that when the sender sends

its message, it gets stored at the right place in the receiver. If the data-structure is not pre-posted, then the receiver will either send a Receiver Not Ready (RNR) packet to the sender or silently drop the message, depending upon the transport service type such as Reliable Connected and Unreliable Connected. In the memory semantic, one of the following is required for message transfer: the sender is made aware of the receiver's buffer address for RDMA Write or the receiver is made aware of the sender's buffer address for RDMA Read. Typically, this information is exchanged via control messages before an RDMA Write is issued by the sender or an RDMA Read is issued by a receiver. Once the peer's buffer address is known, in RDMA Write, the sender initiates a data transfer to the receiver's buffer. Similarly, in RDMA Read, the receiver initiates a data transfer from the sender's buffer to its local buffer.

In InfiniBand, messages are exchanged over channels that connect the endpoint of one application to the endpoint of any other application or service with which the application needs to communicate. These endpoints are termed as **Queue Pairs (QPs)**, consisting of one Send Queue and one Receive Queue. Message transfer operations are performed by adding **Work Requests (WR)** to these queue pairs. An important field in the WR data structure is the operation code (opcode) which should be set according to the desired operation. Another important group of fields are those that require information about the local or remote memory, which should be set according to the operation. Therefore, in order to create WRs, appropriately-sized memory regions have to be first registered with the **Host Channel Adapter (HCA)**. For Send/Receive (non RDMA read/write) operations, WRs have to be posted at both sender and receiver, and the information about their local registered memory regions is specified in the WRs. Several of these non-contiguous local memory regions can be associated with one WR by specifying them in an array of data structure called **Scatter-Gather Element (SGE)**.

When WRs get completed, their completion information is added to a queue called **Completion Queue (CQ)**. InfiniBand provides two ways of knowing completions, *event*

notification and *polling*. By default, event notification is blocking in nature and is *interrupt-driven*. Also, for event notification, a request has to be first made using the appropriate API call. If a completion happens on the associated CQ, then an interrupt is generated based on the request made by the aforementioned API call. So, in interrupt-driven completion detection, the thread looking for completions simply "sleeps" until it is awoken by an interrupt. The idea behind polling based completion detection is very straightforward. Whenever completion information is required, the associated CQ is polled for completion events. Upon success, the API call returns a list of Work Completions (WC), and an empty list is returned if there are no WCs in the CQ. Note that this call is non-blocking. So, it may have to be called multiple times before the completion of a particular WR is known.

2.1.2 NUMA Based Compute Node

In addition to having multiple compute nodes, each compute node of modern clusters may be equipped with multiple CPUs, with each CPU having several cores. Therefore, intra-node communications may happen between CPU cores and between distinct CPU sockets. A NUMA node (Figure 2.3) consists of a CPU and one or more local memory modules that are directly connected to the CPU's memory controller. A CPU can access the memory of other NUMA nodes by means of a proprietary inter-socket interconnect such as Intel's QuickPath Interconnect [38] and AMD's HyperTransport Technology [4]. Data access across NUMA nodes is slower than accessing the local memory but faster than accessing data on another compute node. Inter-socket communication happens implicitly and is entirely managed by the OS. The application merely has to use an inter-process communication (IPC) mechanism such as shared memory.

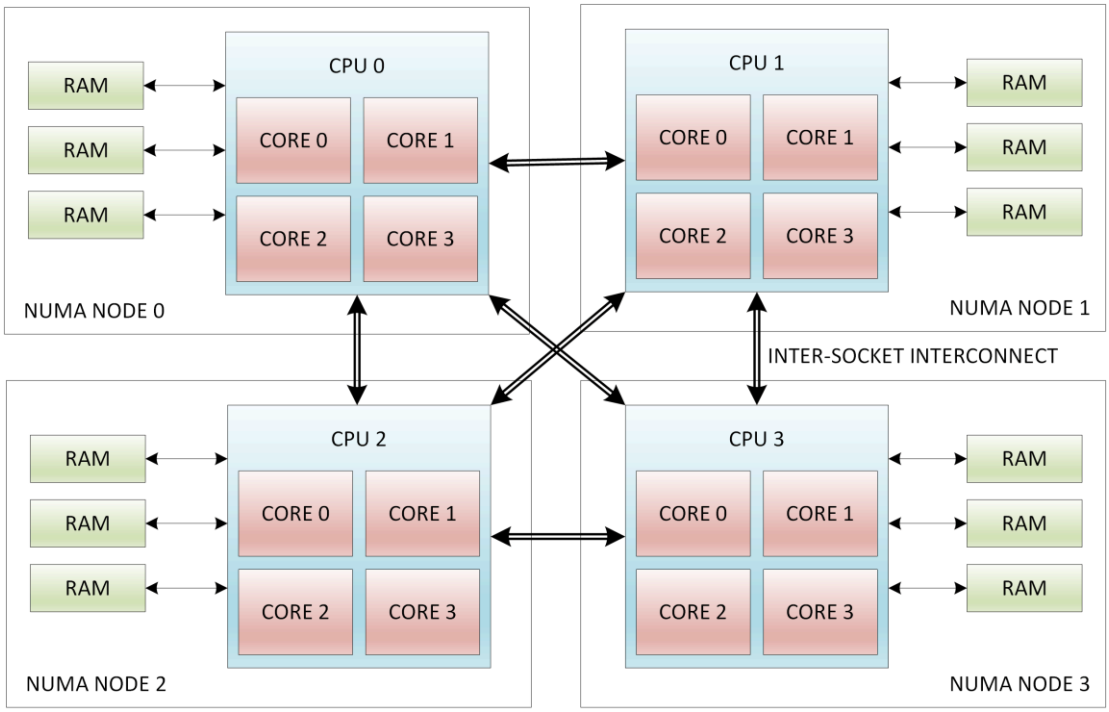


Fig. 2.3. Anatomy of a NUMA Compute Node

2.1.3 Parallel Programming Paradigms

Fast execution of parallel applications not only requires high performance hardware but also the efficient use of the hardware resources. To achieve this and to relieve the application programmer of dealing with low-level communications and synchronizations, several *parallel programming models* have been developed over the years [6]. A parallel programming model essentially provides an abstraction for the underlying hardware and memory architecture to the application programmer. The memory architecture can be shared or distributed, and modern commodity clusters can support either. Nowadays, the parallel programming models can be implemented on any hardware architecture. However, the choice of a particular programming model for a certain application is not trivial, as the same application can produce different performance results when implemented using different programming models [50].

The existing parallel programming models can be broadly classified into three categories: *shared memory*, *partitioned global address space* and *distributed memory* or *message passing*. It is, however, possible to employ multiple parallel programming models in a single application. Such a combination of models is referred to as a *hybrid* model [21]. This chapter provides a brief description about the different parallel programming models and then discusses the different messaging semantics specified in the **Message Passing Interface (MPI)** [54] standard.

2.2 Shared Memory Model

The shared memory model can be thread based or non-thread based. In a non-thread based model, each process has access to a common address space to which reads and writes can be performed asynchronously. To avoid race conditions, access to the common address space can be controlled by the use of locks or semaphores. With respect to the shared memory model, a semaphore is a variable that allows a process to have an exclusive access to the common address space for a specific time interval. During this interval the process that has the exclusive access (lock) can manipulate the data in the common address space; whereas the other processes cannot do so until they are granted the lock. This is a simple programming model with no concept of data ownership; hence, any process can access or manipulate the data in the shared memory. On a machine with a shared memory architecture, support for a purely process based shared memory model may be provided natively through the operating system (OS) or through the compiler. On a distributed memory architecture, such a programing model can be supported through specialized hardware or software, for example, Linda [1] and TreadMarks [41].

In the thread based shared memory programming model, instead of executing instructions on multiple heavy-weight processes, each process can spawn multiple light-weight threads which can execute the tasks concurrently. In this model, all the threads of a particular process are associated with a common address space. If there are multiple processes involved; for example, in case of a

hybrid model, the processes themselves may or may not have a common address space. Two popular standards of this model are POSIX Threads or Pthreads [48] and OpenMP [14].

2.3 Partitioned Global Address Space Model

In the partitioned global address space (PGAS) model [2], the address space of the entire job is global. The global address space is organized in a data structure such as an array or a cube and each element of this data structure is a dataset. In this model, work is performed by threads that collectively act on the global address space. Each thread has affinity with a partition and it works exclusively on that partition. However, it can access data from other partitions. The threads may also have a private address space and may synchronize among themselves by means of barriers and locks. The important distinction between PGAS and the thread based shared memory model is that in the latter, the shared memory is confined locally to the compute node. However, in PGAS, the local memory can be logically shared across the entire cluster. Two widely used implementations of PGAS are Unified Parallel C (UPC) [23] and Coarray Fortran [58].

2.4 Message Passing Model

The message passing model or distributed memory model is the most popular programming model used in HPC [25]. A parallel job executed with this model may consist of several of these processes running on the same physical machine, or span across multiple machines. These processes communicate and synchronize with each other by sending and receiving messages.

The Message Passing Interface [54] is the de-facto standard based on this model. MPICH [55], MVAPICH [56] and OpenMPI [60] are the popular open-source implementations of the MPI standard. These implementations are referred to as middleware as they completely abstract the low-level hardware and communication functions, and expose only the API calls specified in the MPI standard. Most MPI implementations support a variety of networks like InfiniBand [36], iWARP

Ethernet, RoCE and Intel Omni-Path [11]. Also, they are responsible for tasks like process spawning and mapping, setting up connections between the processes and gracefully releasing hardware resources before application termination. The application programmer can simply use the appropriate MPI calls to perform intra-node or inter-node communications.

The MPI standard specifies three types of communication semantics: *two-sided* or *point-to-point communication, collective communication* and *one-sided communication* or *Remote Memory Access (RMA)*.

2.4.1 Point-to-point Communication

As the name suggests, point-to-point communication involves only a source and a destination. The source and destination can be alternatively called **sender** and **receiver,** respectively. The sending of a message is initiated by the **MPI_Send** family of calls and the receiving is initiated by the **MPI_Recv** family of calls. The send requests are matched to the receive requests based on the parameters of **rank**, **tag** and **communicator ID**. The communicator ID or context ID is a handle to a group of processes. Rank specifies the rank of the peer in the specified context ID. The tag parameter can be used to distinguish between multiple messages that involve the same peers in the same communicator. Additionally, the receiver may use wildcards like **MPI_ANY_TAG** and/or **MPI_ANY_SOURCE** to accept messages with any tag and from any source, respectively.

The sending and receiving MPI calls are available in both *blocking* and *non-blocking* variants. A blocking receive call (**MPI_Recv**) will cause the program flow to wait at the call until its expected message has arrived. On the other hand, a blocking send call (**MPI_Send**) returns when the local send buffer used for the communication can be reused. In several implementations, this implies that the message has been sent but it does not guarantee the arrival of the message at the destination. In a non-blocking two-sided MPI call (**MPI_Isend/MPI_Irecv**), a request for the send/receive operation is added to the middleware and the task of blocking is deferred to the

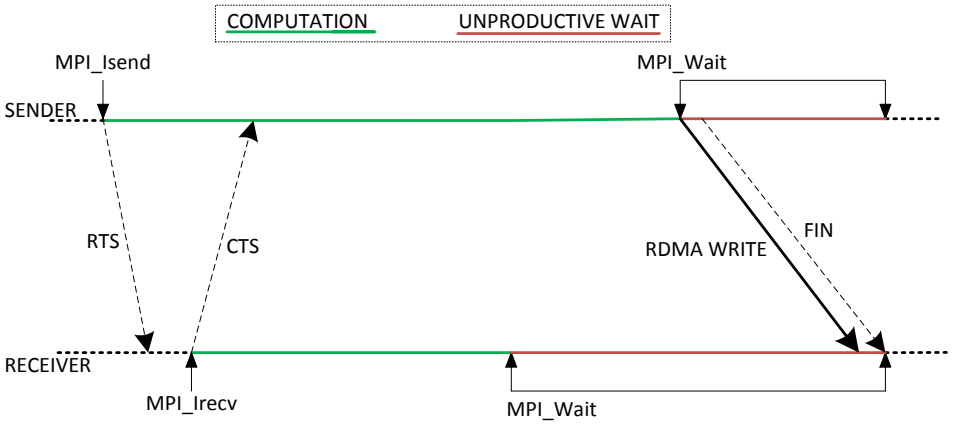
MPI_Wait family of calls. The middleware checks for the completion of this request at each blocking call made for other communications and at **MPI_Test** family of calls. This happens until the communication is complete or until it encounters the request's own MPI_Wait; at which point, the MPI_Wait returns immediately if the communication is complete or blocks until completion. The implementation of the blocking and non-blocking point-to-point MPI calls depends largely on the protocol used for communication.

The eager and *rendezvous* protocols are typically used in the implementations of point-to-point communications. The MPI implementation dynamically decides between these two protocols depending upon the message size. If the message size is below a particular implementation specific threshold then the eager protocol is used; otherwise, the rendezvous protocol is used.

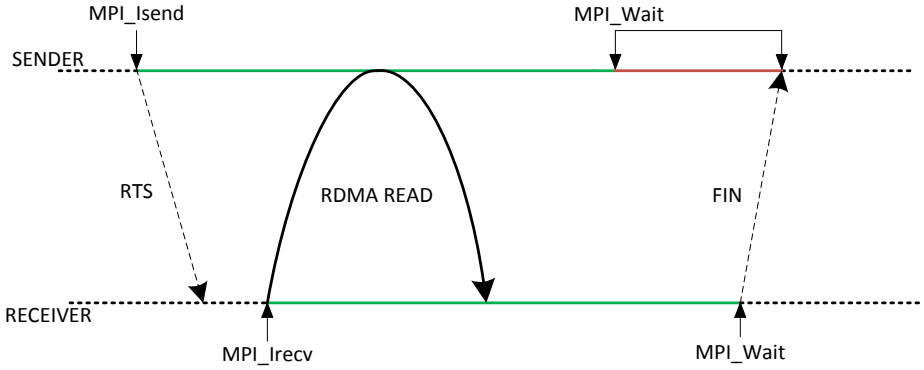
Eager and Rendezvous Protocols

In the eager protocol, the sender does not require any synchronization with the receiver before sending the user data. The rendezvous protocol on the other hand, requires synchronization between the sender and the receiver by means of control/handshaking messages, before the transfer of the user data is initiated. Eager messages are sent through Send/Receive semantics in InfiniBand or through similar mechanisms in other interconnects. In the rendezvous protocol, the control messages are sent eagerly but the application data is transferred through RDMA Read/Write [76].

The rendezvous protocol has traditionally been *sender initiated.* A *receiver initiated* rendezvous protocol is possible but is suboptimal compared to the sender initiated protocol. The receiver initiated rendezvous protocol is discussed in depth in the Chapter 3. Figure 2.4 shows the control signals involved in the sender initiated rendezvous protocol. Figure 2.4(a) shows an RDMA Write based implementation of this protocol. As shown in the figure, the communication is initiated by the sender by sending a **Ready to Send (RTS)** control message to the receiver. This message contains the message matching information (rank, tag and context ID), message size and some other



(a) RDMA Write Based



(b) RDMA Read Based

Fig. 2.4. Sender Initiated Rendezvous Protocols (Sender Arriving First Scenario)

implementation specific information. The message matching information is also referred to as the message envelope. Upon arriving at the matching MPI receive call, the receiver replies by sending a **Clear to Send (CTS)** control message. This CTS control message contains the address of the buffer where the data must be written by the sender. At the MPI_Wait, the sender transfers the user data to the receiver's buffer using RDMA Write and then issues a Finish (FIN) control message to signal the end of the rendezvous communication.

Similarly, Figure 2.4(b) shows an RDMA Read based implementation of the sender initiated rendezvous protocol. As with the RDMA Write based version, the sender initiates the communication by issuing an RTS control message. However, instead of carrying just the message

matching information and message size, this RTS also contains the address of the buffer at the sender where the user data is stored. When the receiver arrives at the matching MPI receive call, it issues an RDMA Read operation to fetch the user data from the remote address specified in the RTS. Finally, at the MPI_Wait call, the receiver signals the end of the rendezvous communication to the sender through a **FIN** control message. Figure 2.4 represents the scenario where the sender arrives first. If the receiver arrives first, then the task of issuing the CTS/RDMA Read is deferred to MPI_Wait.

Message Matching and Message Progression

As previously mentioned, eager messages are sent through InfiniBand's Send/Receive semantics. The registered memory region in this case is not the same as the application buffer, which means that the data sent from the sender to the receiver does not land into its ultimate destination directly but instead to an intermediate location. Figure 2.5 shows the hierarchy of buffers in an MPI process. The data needs to be moved from the application buffer of the sender to the application buffer of the receiver. However, it lands intermediately on the communication buffer. This is required for multiple reasons, such as minimizing memory registration costs and providing support for MPI's message-matching semantics. To ensure that the message gets ultimately delivered to the right location, two important operations are employed, *Message Progression* and *Message Matching*. **Unexpected Message Queue (UMQ)** and **Posted Receive Queue (PRQ)** are two data structures that support these operations. Transferring a message from the application buffer of one peer to the application buffer of the other peer involves several steps. Such steps may require the copying of the message to intermediate buffers at the middleware or to transfer the message through the wire to the other peer. The execution of one or more of these steps is referred to as message progression. The entity that performs the message progression is often

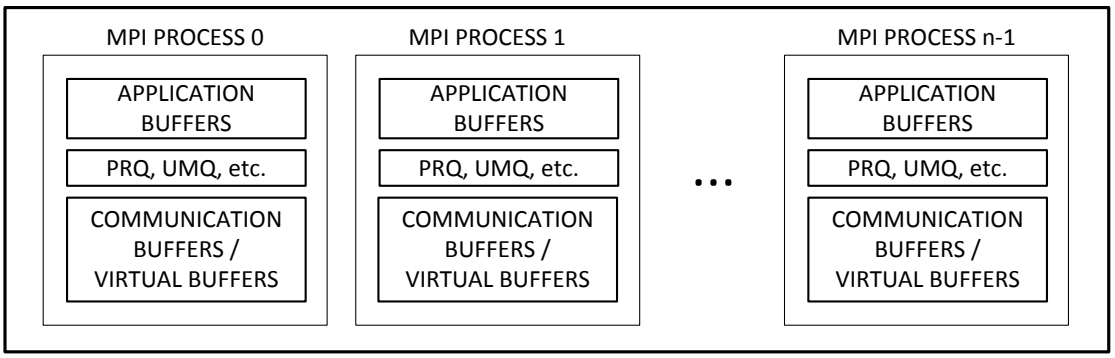


Fig. 2.5. Hierarchy of Buffers in Common MPI Implementations

referred to as the **progress engine**, and invoked by blocking MPI calls, some non-blocking MPI calls and MPI_Test family of calls.

Point-to-point MPI receive calls, expecting a message from another process, first invoke message matching to check if their message has already arrived and processed by the progress engine. This is done by first examining the entries of the UMQ to find a match for the expected message. If a match is found, then the rest of the steps are performed according to the protocol followed. Otherwise, a request object is created for that MPI call and added to the PRQ. Messages and requests are said to be matched if their MPI rank, message tag and context ID conform to each other. To find a match for this request object, the progress engine will be called either immediately or at some point in the future. Upon invocation, the progress engine gets a list of WR completions since its last invocation. Among other things, this list contains information about the receive WRs that were completed. At this point, the arrived messages still reside in the registered memory region (the communication buffer). So, message matching is required for further processing. This is done by first comparing the messages with the request objects in the PRQ. If a match is found then the rest of the steps are performed according to the protocol. Otherwise, an entry is added to the UMQ. Note, that a call to the progress engine will progress all the arrived messages in the registered memory region, regardless of the request that it was called for.

One of the tasks of the progress engine is to check for work completions, which it does by calling the appropriate network API functions. As mentioned earlier, InfiniBand provides two ways of knowing completions, event notification by means of interrupts and polling. Consequently, the progress engine can be based on polling or interrupts. In the interrupt based approach, the progress engine sleeps until an interrupt is generated due to a completion on its associated CQ. The progress engine then returns if it finds the message that it was called for; otherwise, goes back to sleep again. The polling based progress engine, on the other hand, calls the completion detection API function several times in a busy loop until the completion of a particular WR is known.

The choice between polling and interrupts requires a careful consideration of the trade-offs. Polling is more responsive than interrupts but it is not as resource efficient. Since polling requires a continuous examination of the CQs in a busy loop, the progress engine inflicts a 100 percent CPU utilization on the core on which it is mapped. In contrast, an interrupt based progress engine essentially sleeps until a completion, so its CPU utilization during that time is zero. Polling does not require any interaction with the kernel and no context-switching is involved either. However, interrupts are associated with interrupt-generation and context-switching overheads, that make them less responsive compared to polling.

Communication/Computation Overlap

As mentioned earlier, with RDMA, the communication is entirely offloaded to the NIC and the CPU only has to be involved for a very short duration to add a communication request to the NIC. This provides the ability to hide the latency of the communications by overlapping it with the computations. The semantics of blocking point-to-point MPI calls depends a lot on the type of the call (send/receive) and on the protocol used. In general, blocking MPI calls have to wait until the message is progressed partly or completely. This wait becomes even more severe with rendezvous protocols because of the size of the messages and the involvement of control signals. During this

wait, the CPU is basically wasting CPU cycles as the communication is being carried out by the NIC. Also, while the blocking call is waiting for its message to be progressed, the application thread obviously cannot proceed with the computation. Therefore, the use of blocking two-sided calls may cause the communication and computation to get serialized; although, this cannot be avoided in some circumstances. On the other hand, non-blocking calls perform the necessary actions and return immediately. If an expected message is not found then instead of waiting for it, a non-blocking call adds a communication request at the middleware and delegates the task of message progression to the future progress engine calls. After issuing the non-blocking call, the application can immediately proceed to other activities. This promotes communication/computation overlap by deferring the communication to a more opportune time, and sneaking in computation in the meantime. Therefore, the use of non-blocking calls facilitates the overlap of communication and computation.

Figure 2.6 illustrates two code snippets that implement the same logic using blocking and non-blocking receive calls, and compares their communication/computation overlap. In this figure, assume that the message is larger than the eager threshold and that the sender initiated RDMA Read based rendezvous protocol is used. This section aims to explain the concept of communication/ computation overlap in two-sided communication, therefore, it uses a very specific example. For the same reason, this discussion is limited to the receiver side overlap. A detailed discussion on the overlap in other scenarios and other rendezvous protocols can be found in Chapter 3.

The sender and receiver in both the code snippets start with the MPI_Barrier to ensure that line 2 starts at almost the same time in both the peers. Since the focus is on the receiver side overlap, the sender simply issues the non-blocking send call, performs its computation and issues the MPI_Wait call to wait until the communication is complete. The receiver's code in Figure 2.6(a) performs some computation, then issues a blocking receive (MPI_Recv) and continues with its computation. The receiver's code in Figure 2.6(c) performs essentially the same actions but

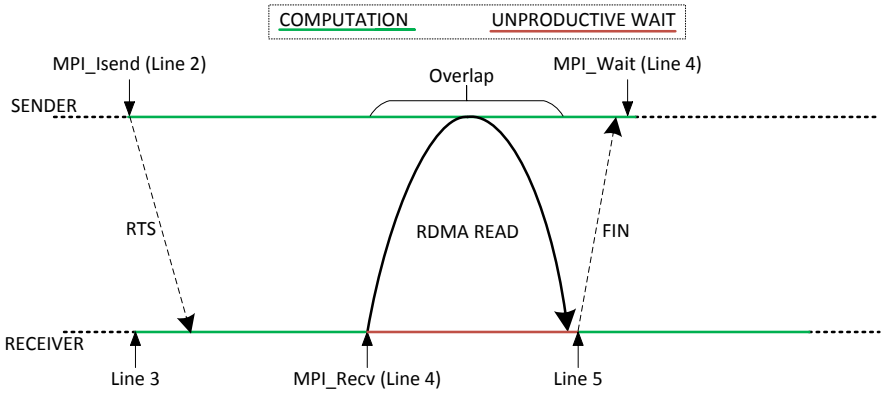
replaces the MPI_Recv with its non-blocking version (MPI_Irecv) and adds an MPI_Wait after the second computation. Note, that in both the receiver snippets, the execution durations of the instrumentation functions at line 2 and line 6/line 7 are negligible compared to the other statements, hence their contribution to the total elapsed time can be ignored. As shown in Figure 2.6(b), the MPI_Recv does not return until the entire message is progressed. This is because it cannot send the Finish (FIN) control message until the RDMA Read is compete. Therefore, the result is a strict serialization of communication and computation. On the other hand, with MPI_Irecv, the RDMA Read overlaps with the computation at line 5. The RTS is already present at the receiver when the MPI_Irecv gets called, so the receiver issues the RDMA Read and continues with its computation. The MPI_Irecv is paired with the MPI_Wait at line 6, which would have ultimately blocked if the communication had not been progressed. In this case, however, the only task left for it is to send the FIN control signal. With same computations and message size, the two code snippets would perform exactly the same task, however, the lack of overlap with the blocking receive would cause the elapsed time to be greater.

2.4.2 Collective Communications

Along with performing communications between pairs of MPI processes, the MPI standard provides the provision of involving multiple processes in a communication through MPI collectives. The standard specifies collectives for communications such as: sending the same data from one process to the others (**MPI_Bcast**), distributing chunks of data from one process to the others (**MPI_Scatter**), gathering data from different processes to a single process (**MPI_Gather**), performing a reduction operation on numeric data supplied by the participating processes (**MPI_Reduce**), etc. Similar to point-to-point MPI calls, collectives are available in both blocking and non-blocking variants. At the middleware, collectives may be implemented using two-sided MPI calls [77] or one-sided MPI calls [79], or using special network API calls that are optimized

SENDER:	RECEIVER:
1. MPI_Barrier(MPI_COMM_WORLD) 2. MPI_Isend 3. Computation 4. MPI_Wait	1. MPI_Barrier(MPI_COMM_WORLD) 2. Measure start_time 3. Computation 4. MPI_Recv 5. Computation 6. Meaure stop_time 7. elapsed_time = stop_time – start_time

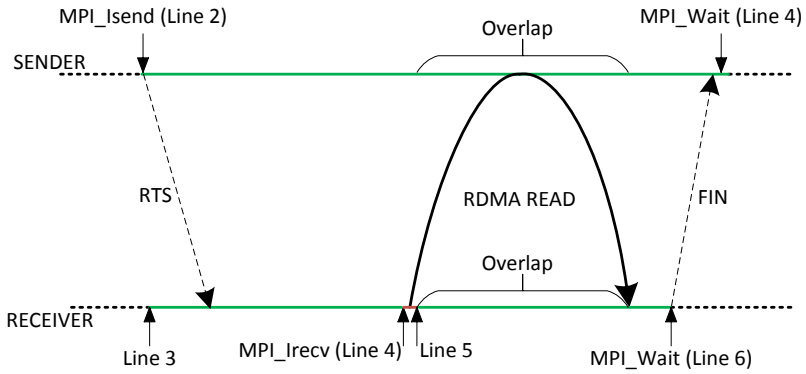
(a) Code Snippet with Blocking Receive Call (MPI_Recv)



(b) Timing Diagram for (a)

SENDER:	RECEIVER:
1. MPI_Barrier(MPI_COMM_WORLD) 2. MPI_Isend 3. Computation 4. MPI_Wait	1. MPI_Barrier(MPI_COMM_WORLD) 2. Measure start_time 3. Computation 4. MPI_Irecv 5. Computation 6. MPI_Wait 7. Meaure stop_time 8. elapsed_time = stop_time – start_time

(c) Code Snippet with Non-Blocking Receive Call (MPI_Irecv + MPI_Wait)



(d) Timing Diagram for (c)

Fig. 2.6. Comparison of Communication/Computation Overlap between Blocking and Non-Blocking Two-Sided Calls in RDMA Read Rendezvous Protocol

for collectives [32, 37]. Therefore, the message progression semantics of a collective communication would be similar to that of the implementation on which it is based.

2.4.3 One-Sided Communication

One-sided communication was introduced in the version 2.0 of the MPI standard and has undergone significant revisions in version 2.2 and version 3.0. It is not as widely used in scientific applications as two-sided communication [15], but is known to have lower latencies for large messages [40]. Unlike two-sided communication, in one-sided communication, data can be moved to or fetched from a remote process without requiring any synchronization with the remote process. Also, the remote process does not have to issue a matching API call to initiate the data transfer, unlike two-sided communication. Communications are done by directly performing operations on the exposed region of the remote process's memory. Hence, one-sided communication is also referred to as *Remote Memory Access (RMA)*. In RMA terminology, the exposed remote buffer is called a **window**. The remote process is called the **target** and the process that performs operations on the window of the target is called the **origin**. Before starting the RMA operations, all the involved peers call the **MPI_Win_create** function. MPI_Win_create is a collective call that returns a window object which can be used by these processes to perform RMA operations. Each process may specify a window of its local memory that it intends to expose to RMA accesses by the other processes in the group. Alternatively, a process may elect to expose no memory by specifying a window size of zero.

The MPI standard specifies three types of RMA operations: **MPI_Put** to transfer data from the origin to the target, **MPI_Get** to transfer data from the target to the origin and **MPI_Accumulate** to perform a remote arithmetic operation. These operations do not require target-origin synchronizations; however, RMA as a whole is not synchronization free. In MPI, the RMA synchronizations can be of the following two types, namely, *active target* and *passive target*.

Active Target Synchronization

In active target, synchronization between the targets and origins is required at two stages. The duration between these stages is the period for which a window remains exposed for RMA operations, and is referred to as the *epoch*. In the first stage, an epoch starting call is required among the peers to exchange window exposure information. Then, after all RMA operations are performed on the window, an epoch completing function is called to signal that the window is no longer required to be exposed. A target window can be accessed by RMA operations only within an *exposure epoch*. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Similarly, RMA synchronization calls are executed by the origin to start and complete an *access epoch*, during which it may issue RMA operations to the target's window.

There are two semantics by which active target synchronization happens, namely, *Fence* and *General Active Target Synchronization (GATS)*. Figure 2.7(a) shows a code snippet that uses fence synchronization for RMA communications. In this, all processes associated with the window call **MPI_WIN_FENCE** to start an epoch which is both an access epoch as well as an exposure epoch. Consequently, all the peers are simultaneously both a target and an origin. If an origin issues an RMA operation during the exposure epoch of the peer then the operation is performed immediately. If an RMA operation is issued earlier than the exposure epoch, then the operation is queued as a request in the middleware of the target, which is processed when the origin exposes its window. Once all the RMA operations are done, each peer calls MPI_WIN_FENCE again to synchronize and complete the access and exposure epochs.

Unlike fence, in GATS (Figure 2.7(b)), the epoch opening synchronization can either be for an exposure epoch or an access epoch. Therefore, during an epoch, a process can either be an origin or a target, but not both. In Figure 2.7(b), Process0 and Process2 are origins which start the access epoch by calling **MPI_WIN_START**. In this call, they specify the window and the process that

Process0	Process1	Process2
MPI_WIN_FENCE(win)	**MPI_WIN_FENCE(win)**	**MPI_WIN_FENCE(win)**
MPI_GET(win,1)	MPI_PUT(win,0)	MPI_PUT(win,0)
MPI_PUT(win,2)	MPI_PUT(win,2)	MPI_GET(win,1)
..	MPI_GET(win,0)	..
..	..	..
MPI_WIN_FENCE(win)	**MPI_WIN_FENCE(win)**	**MPI_WIN_FENCE(win)**

(a) Fence Epoch

Process0	Process1	Process2
	MPI_WIN_POST(win, {0,2})	
MPI_WIN_START(win, {1})		**MPI_WIN_START(win, {1})**
MPI_GET(win,1)		MPI_PUT(win,1)
MPI_PUT(win,1)		MPI_GET(win,1)
..		..
MPI_WIN_COMPLETE(win, {1})		**MPI_WIN_COMPLETE(win, {1})**
	MPI_WIN_WAIT(win)	

(b) General Active Target Synchronization (GATS) Epoch

Process0	Process1	Process2
MPI_WIN_LOCK(win, 1)		**MPI_WIN_LOCK(win, 1)**
MPI_GET(win,1)		MPI_PUT(win,1)
MPI_PUT(win,1)		MPI_GET(win,1)
..		..
MPI_WIN_UNLOCK(win, 1)		**MPI_WIN_UNLOCK(win, 1)**

(c) Lock/Unlock Epoch

Figure 2.7: MPI 3.1 RMA Synchronizations

will be their target for RMA operations. In this example, Process1 is the target. Process1 starts the exposure epoch by calling **MPI_WIN_POST** and specifying the processes that are allowed to operate on its window. Once all the RMA calls are made, the origins complete the access epoch by issuing the **MPI_WIN_COMPLETE** synchronization call. The target calls **MPI_WIN_WAIT** to wait until all its origins have executed the MPI_WIN_COMPLETE function. After which, the exposure epoch completes. The MPI standard does not specify the blocking or non-blocking behaviour of the synchronization calls. In common MPI implementations, for both Fence and

GATS, the epoch starting synchronization calls are non-blocking and the epoch completing ones are blocking.

Passive Target Synchronization

In passive target synchronization, the target (Process1 in Figure 2.7(c)) does not make any synchronization calls. In fact, it does not even need to be aware of RMA operations upon it. Passive target synchronization can be thought of as an emulation of distributed shared memory. To perform RMA operations on the target, an origin (Process0 or Process2 in Figure 2.7(c)) needs to first open an epoch by calling the **MPI_WIN_LOCK** function. This lock can be *exclusive* or be *shared* with other origins. If an origin has an exclusive lock on a target, then other origins cannot lock that target until the lock holding origin completes the epoch by issuing the **MPI_WIN_UNLOCK** call. However, in a shared lock synchronization, a target can be locked and operated upon by multiple origins at once. It is also possible to lock/unlock all the processes associated with an RMA window using a pair of **MPI_WIN_LOCK_ALL** and **MPI_WIN_UNLOCK_ALL** calls, but these calls can only request shared locks.

Message Progression

Message progression in one-sided communication is required for the epoch opening and closing signals, and for the RMA operations. These two operations are similar to the progression of the control signals and the message in the rendezvous protocols, with the important distinction that multiple messages can be transferred within each epoch in case of one-sided communication. On RDMA enabled networks, the RMA operations are performed using RDMA Read or RDMA Write. As mentioned earlier, the MPI standard does not define the blocking or non-blocking nature of the epoch manipulation calls and the designers are free to decide this behaviour for their MPI implementation. However, from version 3.0, the MPI standard mandates a non-blocking behavior

for all RMA operations. Depending upon whether the synchronization calls are blocking or non-blocking, the RMA operations are progressed in one of the following ways:

a) With blocking synchronization calls, any RMA operation that gets issued is progressed immediately using RDMA. In case of fence and GATS, blocking synchronization guarantees that the exposure epoch will be open when the RMA operation is issued. Similarly, in case of exclusive lock epoch, blocking synchronizations ensure that the lock to the target window will already be acquired when the RMA operation call is made. Therefore, there is no necessity of queuing or deferring the RMA operations.

b) With non-blocking synchronizations, in case of fence and GATS, the RMA operation is progressed immediately if the exposure epoch is open. Similarly, in case of exclusive lock epoch, the RMA operation will be progressed immediately if the lock to the target window is already acquired. However, if the exposure epoch is found to be closed or if the lock is not acquired yet then the RMA operation is queued at the origin's middleware and deferred to be progressed later. This message will then be attempted to progress during subsequent calls to the progress engine, and the RMA operation will be initiated when the transfer conditions are met, that is, when the synchronizations are complete. Once this happens, subsequent RMA operations to the same target will always be progressed immediately.

Note, that in shared lock epochs, all RMA operation calls are progressed immediately as the origin is not required to wait for the exposure epoch or the lock.

In fence and GATS, the epoch opening synchronization requires the targets to signal the opening of their exposure epochs to all the origins associated with the window object. Essentially, all the targets act as the sender and all the origins act as the receivers. In the epoch closing synchronization, the roles get reversed and the origins signal the targets about the closing of the access epoch. In case of exclusive lock epoch, the target does not issue any synchronization calls but participates by acting as the mediator between the other origins. It does this implicitly at the

middleware when the progress engine is invoked. When the origins want the lock to the target's window, they send a signal to the target to request for the lock. The lock is granted by the progress engine of the target by signalling the origin whose request arrived earliest at the target. After completing all its RMA operations, the origin that has the lock has to relinquish it by signalling the target. When the next call to the progress engine at the target sees this signal, it will grant the lock to the origin that is next in the queue.

The semantics of blocking RMA synchronizations are very similar to that of MPI_Barrier. A blocking RMA synchronization call does not return until all the peers associated with the window object have issued the matching epoch manipulating call. On the other hand, a non-blocking synchronization call performs the necessary signalling to the peers if required and returns immediately. RMA synchronizations may be implemented using two-sided communication with the eager protocol or using RDMA directly. If the implementation is done using the eager protocol, then MPI send/receive calls are used at the middleware for signalling. Therefore, the message matching and message progression semantics of the RMA synchronizations in this case would be exactly the same as two-sided communication. The other option is to use RDMA Write and directly write the signal to a pre-allocated remote buffer. Message matching would not be required in this case and the progression will happen immediately as the signal is directly transferred to the intended location. However, this would require an exchange of the remote addresses among the peers before the synchronization calls are made.

Communication/Computation Overlap

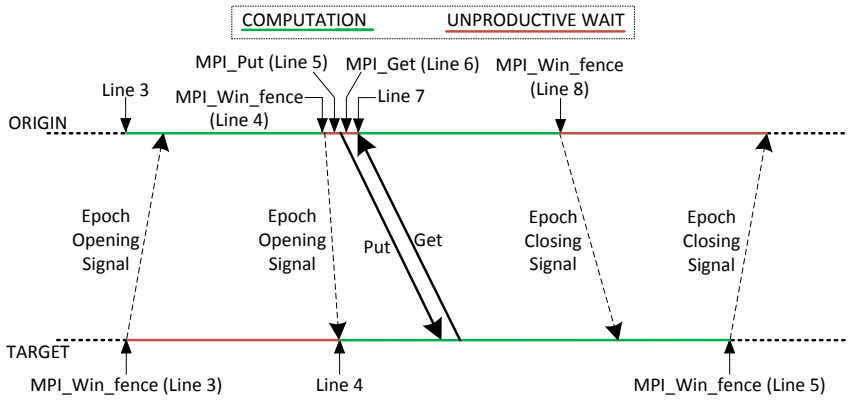
The discussion in Section 2.3.1 about the need of non-blocking calls for communication/ computation overlap holds true for one-sided communication as well. However, it is important to note that with blocking epoch manipulating calls, the RMA operations are progressed immediately, so there will always be some degree of overlap. The major problem with blocking calls is

unproductive wait which can easily get propagated to other peers. These unproductive waits are caused when a blocking epoch manipulating call has to wait for some signal from its peers. This signal could be about the opening or closing of an epoch, or about the granting or revoking of a lock. Such waits are unproductive because while the call is blocking, the CPU is wasting its cycles which could have been used for computation. A non-blocking epoch manipulating call never waits for the arrival of its signal. Therefore, it can completely avoid an unproductive wait. This does not mean that it makes the entire system wait-free. RMA communications must be ultimately blocked somewhere to make sure that the pending RMA operations are progressed. Because of the impact that it has on the performance of RMA communications, the notion of overlap in this section is equivalent to the absence of unproductive wait.

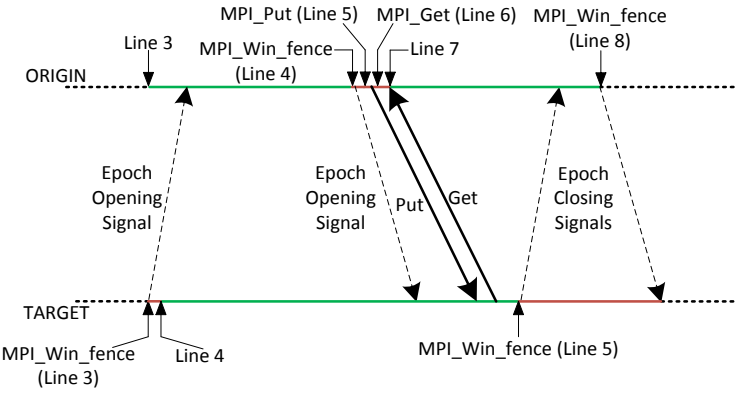
Figure 2.8 shows a code snippet that uses fence epochs and compares the timing diagrams of this snippet with blocking and non-blocking epoch manipulating calls. The objective is to introduce the concept of unproductive waits in RMA by using blocking and non-blocking fence as the example. However, such inefficiencies can be found in other types of RMA synchronizations as well, the details of which are covered in Section 3.2.1. Figure 2.8(b) uses blocking calls for epoch opening synchronization as well as for epoch closing. Figure 2.8(c), on the other hand, uses non-blocking calls for epoch opening synchronization, but the epoch closing synchronization is performed using blocking calls. This is because of the previously stated reason that RMA communications must be ultimately blocked somewhere. As can be seen in Figure 2.8(b), the target's fence arrives earlier but it cannot return until it receives the epoch opening signal from the origin. This causes it to unnecessarily wait for the entire duration of computation that is performed at line 3 at the origin. This unproductive wait can be observed between line 3 and line 4 at the target in Figure 2.8(b). On the other hand, in Figure 2.8(c), the non-blocking fence call at line 3 of the target initiates an epoch opening signal to the origin and returns without waiting for the origin's signal. This enables the target to avoid the wastage of CPU cycles by immediately preceding to the

ORIGIN:	TARGET:
1. MPI_Barrier 2. Measure start_time 3. Computation 4. MPI_Win_fence 5. MPI_Put 6. MPI_Get 7. Computation 8. MPI_Win_fence 9. Meaure stop_time 10. elapsed_time = stop_time – start_time	1. MPI_Barrier 2. Measure start_time 3. MPI_Win_fence 4. Computation 5. MPI_Win_fence 6. Meaure stop_time 7. elapsed_time = stop_time – start_time

(a) Code Snippet for RMA Communications with Fence Synchronization



(b) Timing Diagram with Blocking Fence Synchronizations



(c) Timing Diagram with Non-Blocking Epoch Opening Fence Synchronization

Fig. 2.8. Comparison of Unproductive Waits with Blocking and Non-Blocking RMA Synchronization Calls

computation at line 4. Similarly, the epoch opening call at the origin initiates a signal to the target and returns. When the RMA operation calls at line 5 and line 6 of the origin are issued, the

information about the opening of the target's exposure epoch is already present, so those RMA operations are progressed immediately. Keeping the computations and message sizes same, the elapsed time at the target would be greater with the blocking fence call, and the difference would be equivalent to the computation duration at line 3 of the origin.

2.5 Summary

This chapter provides the relevant background for the rest of the content in this thesis. It starts by providing an insight on the modern HPC systems by describing a commodity cluster. It highlights the different components of a compute node, emphasizes the importance of high performance interconnects, introduces RDMA and InfiniBand, and discusses intra-node and inter-node communications.

In HPC, software is as critical to the performance as hardware, therefore, the above discussion is followed by a discussion on different parallel programming paradigms such as shared memory, PGAS and message passing. This thesis is focused on the MPI standard, therefore, the rest of this chapter provides an in-depth background on the different messaging semantics of MPI and discusses their communication/computation overlap. The MPI standard specifies three ways of exchanging messages: two-sided communication, collective communication and one-sided communication.

This chapter describes important aspects of point-to-point communications, such as the eager and rendezvous protocols, message matching and message progression in the MPI middleware and the concept of communication/computation overlap in two-sided communications. Then, it briefly describes the collective communications and mentions the different ways that they may be implemented. Finally, this chapter discusses the different aspects of one-side communication, such as active and passive RMA synchronizations, RMA operations, message progression and communication/computation overlap.

Chapter 3 discusses the inefficiencies associated with the massage passing semantics and their impact on the performance of parallel applications. Also, it presents a literature review of the approaches that are aimed at addressing the inefficiencies.

Chapter 3

Literature Review

As discussed in Chapter 2, increasing the parallelization of applications may lead to a diminishing return on performance because of the increased need for communications. Therefore, in order to achieve appreciable speedup, it becomes important to hide the communication latencies through communication/computation overlap. The idea behind this is to let the NIC perform the communications with other NICs and switches, without engaging the CPU for the entire length of a particular communication. Reduced utilization of the CPU for communication translates to its greater availability for computation, which ultimately leads to a reduced application execution time. This is possible in RDMA enabled networks, which offer zero-copy and OS-bypass mechanisms. In such a network, the NIC has the ability to transfer the data from one host to the other, without having to perform any intermediate copies and without involving the operating system of either of the hosts. However, this requires an effective use of network APIs. In MPI's context, overlap can be supported at the application layer by careful usage of non-blocking communication calls, such that long communications can happen while the application is busy in a computation. This practice is often referred to as *latency hiding*. This chapter investigates the first research question mentioned in Section 1.2. It discusses the application layer inefficiencies that may lead to the serialization of computation and communication; and then discusses the research proposals that aim to improve the overlap by suggesting modifications at the MPI middleware or at the network layer.

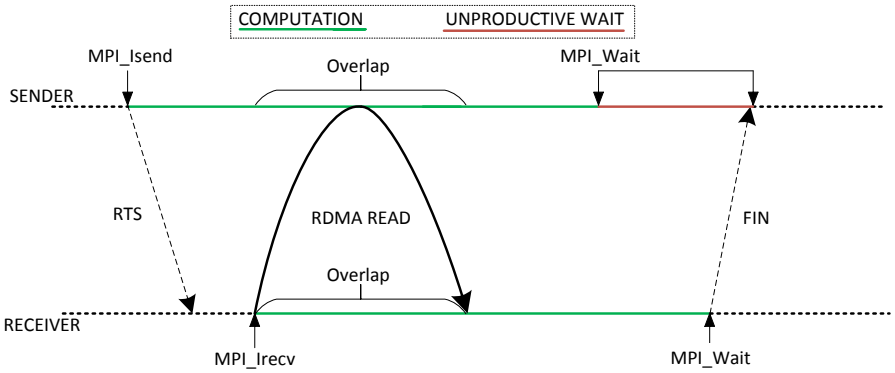
3.1 Point-to-Point Communication

3.1.1 Inefficiencies Associated with the Rendezvous Protocols

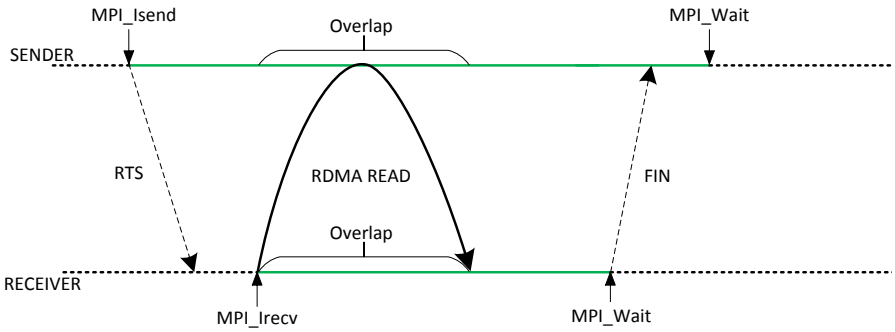
As previously mentioned, the rendezvous protocol has traditionally been sender initiated. The details of the receiver initiated rendezvous protocol will be discussed shortly in Section 3.2. The sender-initiated rendezvous protocol can employ either an RDMA Read based strategy (Figure 3.1) or an RDMA Write based strategy (Figure 3.2). Also, recall that the control messages are sent through the eager protocol. Figure 3.1 and Figure 3.2 show the signaling, data movement and the MPI_WAIT duration under the following scenarios of arrival orders of the sender and the receiver:

- Sender arrives first and calls its MPI_Wait ahead of the receiver's MPI_Wait.
- Sender arrives first and calls its MPI_Wait after the receiver's MPI_Wait.
- Receiver arrives first and calls its MPI_Wait ahead of the sender's MPI_Wait.
- Receiver arrives first and calls its MPI_Wait after the sender's MPI_Wait.

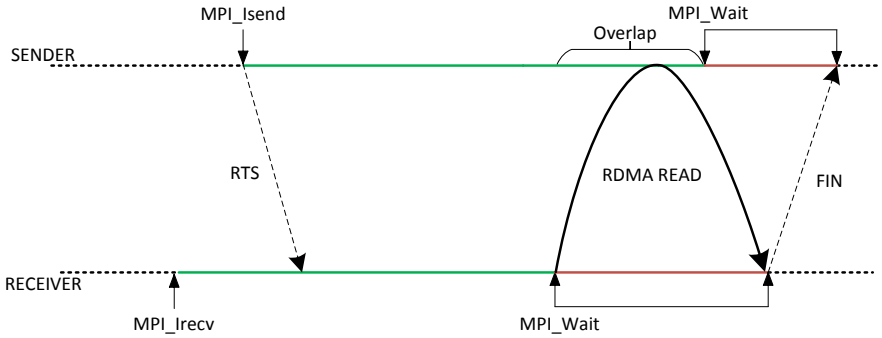
In the rendezvous protocol, a non-timely arrival of the control messages may cause the peers to waste valuable CPU cycles at the MPI_Wait. This unproductive wait may be caused due to a delayed arrival of a control message or a combination of a late control message along with a synchronous message propagation. For instance, consider Figure 3.1(c) and Figure 3.1(d), where the MPI_Isend call gets issued later than its MPI_Irecv call. When the receiver issues the MPI_Irecv call, it does not find its control signal (RTS). This RTS will contain the address of the remote buffer so the receiver cannot initiate an RDMA Read and must return immediately. After returning, the receiver enters into a long computation, during which the RTS arrives. Even though the RTS has arrived at the receiver, the RDMA Read for the MPI_Irecv cannot be issued as the receiving application is busy in the computation. The computations at both the peers end and they issue the MPI_Wait call. Both of these MPI_Wait calls must block until the transfer of data and control signals are complete. The receiving side blocks at its MPI_Wait until the RDMA Read is complete



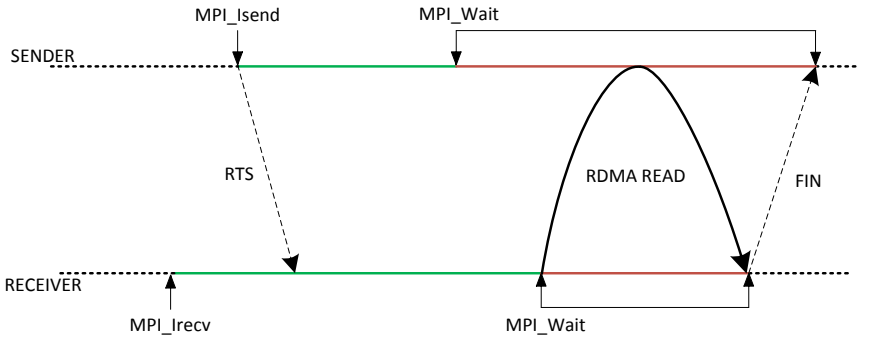
(a) Sender Arrives First and Issues MPI_Wait First



(b) Sender Arrives First, Receiver Issues MPI_Wait First

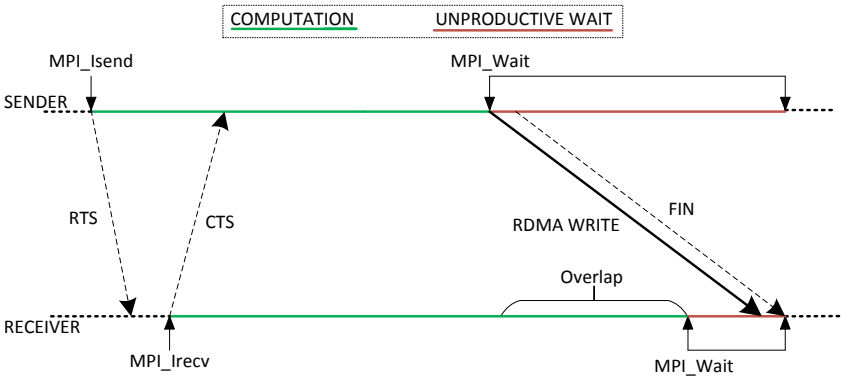


(c) Receiver Arrives First and Issues MPI_Wait First

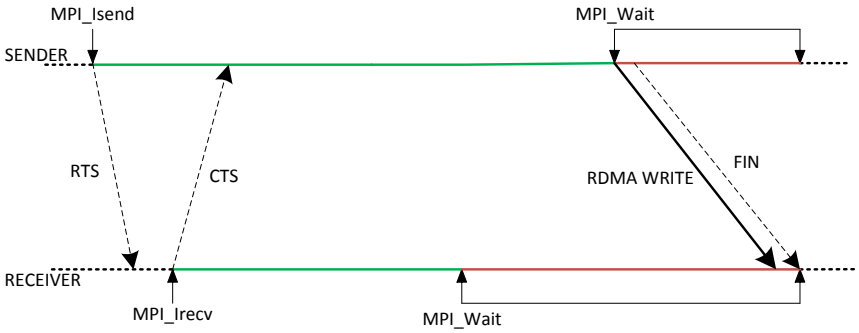


(d) Receiver Arrives First, Sender Issues MPI_Wait First

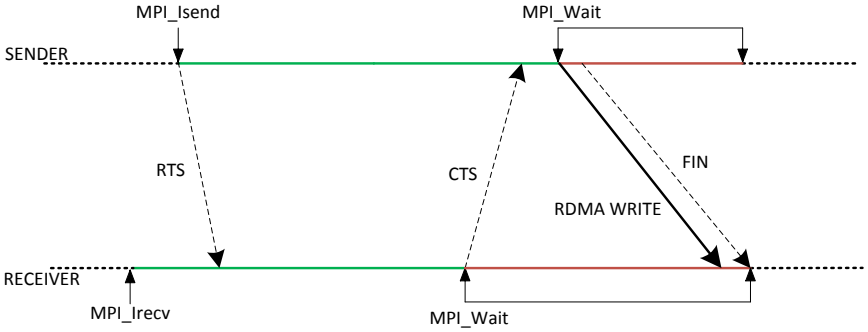
Fig. 3.1: Sender Initiated RDMA Read Based Rendezvous Protocols



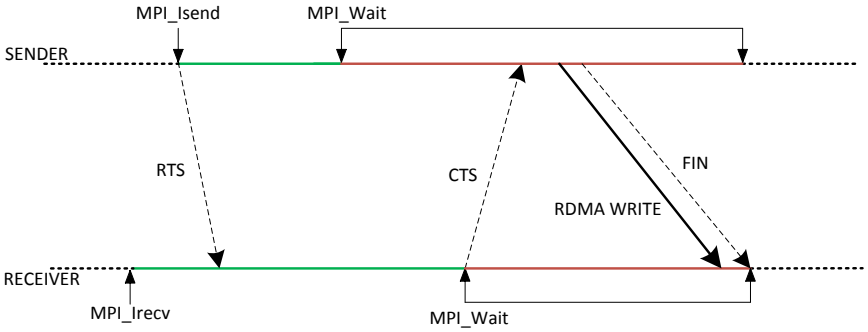
(a) Sender Arrives First and Issues MPI_Wait First



(b) Sender Arrives First, Receiver Issues MPI_Wait First



(c) Receiver Arrives First and Issues MPI_Wait First



(d) Receiver Arrives First, Sender Issues MPI_Wait First

Fig. 3.2: Sender Initiated RDMA Write Based Rendezvous Protocols

and returns after it has sent the FIN control signal. The delay at the sender's MPI_Wait depends upon the arrival of the FIN control message, which could be large if the FIN is very late or negligible if the MPI_Wait gets issued after the arrival of the FIN control message. These delays at the MPI_Waits of the sender and receiver are essentially unproductive because with RDMA, the CPU does not have to be actively involved in the computation. This leads to the wastage of CPU cycles which could have been used productively for the computations. On the other hand, as shown in Figure 3.1(a) and Figure 3.1(b), a timely arrival of the RTS would have overlapped the RDMA Read with the computation and led to negligible unproductive waits.

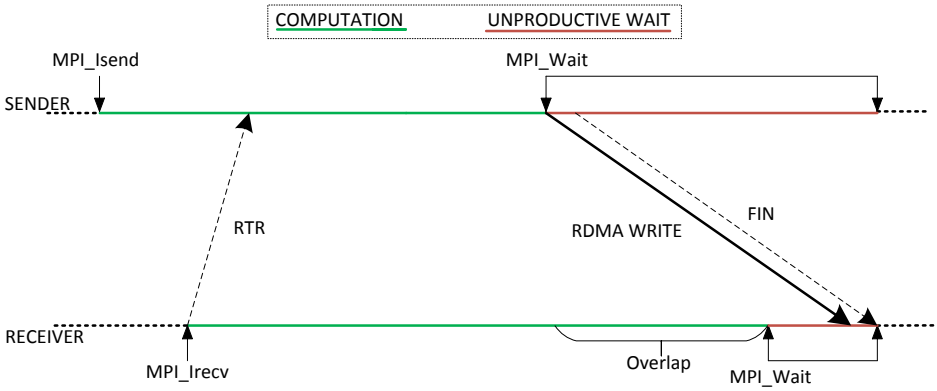
In the RDMA write based protocol (Figure 3.2), the receiver conveys the address of the remote buffer to the sender through the CTS control signal. Since this is a sender-initiated protocol, it cannot do so until it receives the RTS from the sender. Compared to the RDMA Read based protocol, the first problem this leads to is the requirement of an extra control signal. Moreover, there is no scenario of arrival orders in which there can be a natural communication/computation overlap for both the peers. The reason behind this is that, regardless of the arrival orders of MPI_Isend and MPI_Irecv, it is impossible for the MPI_Isend to have complete message progression information when it is issued. A receiver side overlap is possible if the sender arrives ahead of the receiver and the receiver issues its MPI_Wait call substantially later than its peer (Figure 3.2(a)). Overall, there is only one scenario (Figure 3.1(b)) in which both the sender and the receiver can be wait free. This happens when the RTS arrives ahead of the arrival of the receiver and the receiver's MPI_Wait arrives ahead of the sender's MPI_Wait, so that the sender does not have to wait for the FIN control message. In general, in a sender initiated RDMA Read based protocol, a good degree of overlap can be expected for both the peers if the sender arrives ahead of the receiver (Figure 3.1(a) and Figure 3.1(b)).

3.1.2 Literature Review

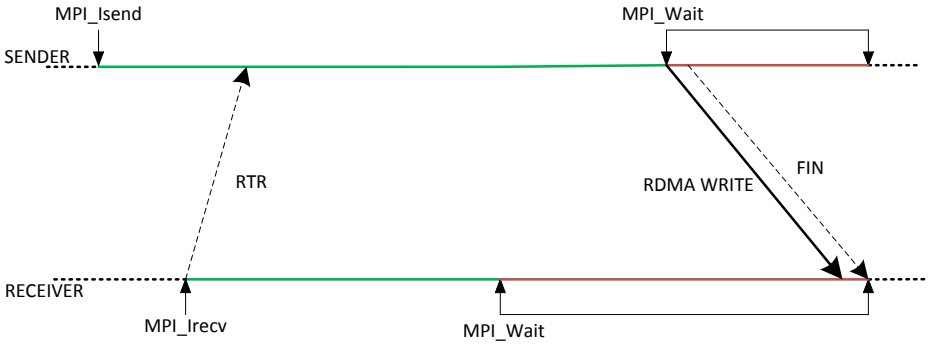
There have been several research proposals over the years to improve communication-computation overlap for point-to-point communication. Particularly, the ones related to the middleware and the network layer can be classified into three categories: *protocol improvement* [66, 75, 51], *hardware-assisted approaches* [43, 27, 62] and *host-based approaches* [31, 46, 73]. Protocol improvement methodologies involve either a redesign of the handshaking mechanisms or adaptive/predictive switching between the protocols. Hardware-assisted approaches make use of special NICs that support offloading of communication primitives. Host-based approaches use CPU or accelerator cores for overlap.

Protocol Improvement Approaches

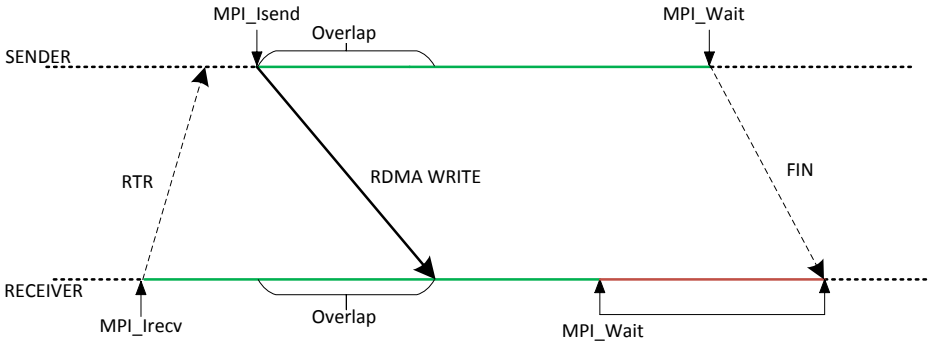
In [61, 66], the authors describe a receiver-initiated rendezvous protocol instead of the traditional sender-initiated protocol that we have discussed thus far. In [61], this scheme is used in a light-weight implementation of MPI for the Cell Broadband Engine [30]. Figure 3.3 shows the timing of control and data messages under different arrival orders of MPI calls. In this protocol, the RTS is replaced by the **Request to Receive (RTR)** control signal. This signal carries the remote address of the buffer at the receiver to which the RDMA Write will be performed by the sender. Figure 3.3(d) shows the ideal situation where a full overlap is possible at both the sender and the receiver. In this case, the RTR is already present at the sender when the MPI_Isend call is issued. So it can immediately progress the message by RDMA Writing it to the remote buffer. If there is enough computation at the sender, then the entire message will be overlapped and the sender's MPI_Wait will simply have to send the FIN control message without waiting. Also, since the FIN has already arrived at the receiver, its MPI_Wait does not have to block either. In general, the sender can be wait free if the receiver is early (Figure 3.3(c) and Figure 3.3(d)). Also, as shown in Figure 3.3(a), a receive side overlap is possible if the receiver's MPI_Wait call is sufficiently late



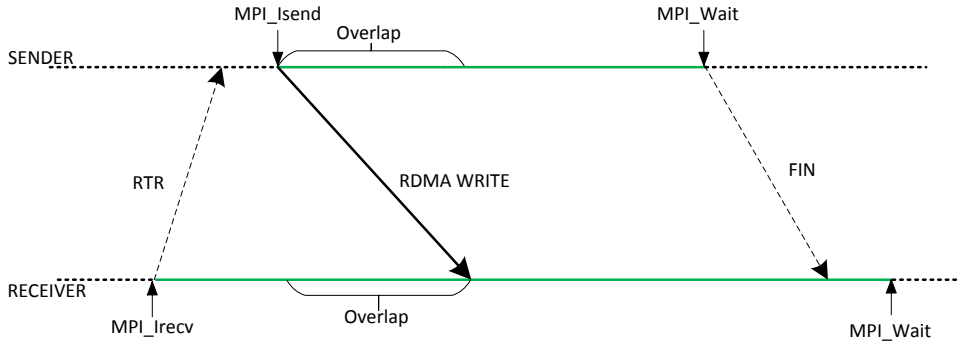
(a) Sender Arrives First and Issues MPI_Wait First



(b) Sender Arrives First, Receiver Issues MPI_Wait First



(c) Receiver Arrives First and Issues MPI_Wait First



(d) Receiver Arrives First, Sender Issues MPI_Wait First

Fig. 3.3: Receiver Initiated RDMA Write Based Rendezvous Protocols

compared to the sender's MPI_Wait call. In fact, a full receive-side overlap is possible if its MPI_Wait is called after the arrival of the FIN control message. Receiver-initiated rendezvous protocol can be implemented with both RDMA Read and RDMA Write. However, using RDMA Read is suboptimal as it entails the usage of an extra control signal and loses the natural overlap ability of RDMA Write when the receiver arrives first.

In [61], the authors argue that this approach is well suited for small-memory processors such as the Cell Synergistic Processing Elements (SPE), which run applications that have a regular communication pattern. However, this approach cannot efficiently deal with two specifications in the MPI standard. First, the MPI standard specifies a "push" communication mechanism, where the receiver may choose to accept messages from an arbitrary sender using the MPI_ANY_SOURCE wildcard. However, the sender has to be specific. Because of this, the receiver-initiated protocols cannot cope with wildcards effectively as it would be impractical for the receiver to send RTRs to all the processes in the communicator. Second, according to the MPI standard, the receiver may specify a buffer size that is larger than the data that it receives. However, the protocol that ends up being used depends on the exact size of the message, which is known only to the sender. If the size of the buffer at the receiver is larger than the eager threshold and the size of the data is less than the threshold, then the receiver will incorrectly issue an RTR for a message that ends up being sent eagerly.

A receiver-initiated rendezvous protocol is also employed in Gravel [16], which is a communication library that can be used in MPI applications to replace some MPI calls with Gravel calls. Using the APIs in this library, the programmer can explicitly specify the protocol to be used for a particular communication. [66] presents a mechanism to *adaptively* select between an RDMA Read based sender-initiated protocol or an RDMA Write based receiver-initiated protocol based on the arrival order of the communicating processes. If the sender arrives first, then the RDMA Read based sender-initiated rendezvous protocol is used, otherwise, the RDMA Write based receiver-

initiated protocol is used. This approach assures overlap when there is a sufficient amount of delay between the arrival of the sender and the receiver.

In [74, 75], in addition to adaptively selecting between the protocols, a set of rendezvous protocols are proposed in which the sender buffers the message locally if the receiver is very late. Specifically, if the sender arrives at its MPI_Wait and the receiver has not arrived at its MPI_Irecv yet, then the sender buffers the message locally, informs the receiver of the protocol change and exits. This enables the sender to be wait-free if the receiver is late. Upon arriving at the MPI_Irecv, the receiver can issue an RDMA Read to fetch the data from the sender. Similarly, in [51] this buffering is done at the receiver by RDMA writing the message to a pre-allocated buffer. When the receiver arrives, it can fetch its message from this buffer. These buffering approaches, aim to improve the sender side overlap. However, they pose a risk of *memory exhaustion* [12], especially considering the relatively larger size of the rendezvous messages as compared to the eager messages.

Adaptive approaches employing the receiver-initiated protocols, suffer from the inabilities of the receiver-initiated protocols which are discussed above. In addition, none of these protocols cope well with a scenario called *similar arrival times* [75]. This situation happens when the sender arrives at its MPI_Isend at almost the same time as the receiver arrives at its MPI_Irecv. The RTR may be on its way but the sender does not see it, so it mistakenly sends an RTS to the receiver. Similarly, the receiver mistakenly sends an RTR because it is not aware of the incoming RTS. Hence, this situation leads a race condition.

This race condition can be avoided by using additional synchronization steps [66, 75] or by employing *predictive* mechanisms [51, 75]. Extra synchronization steps limit the overlap potential and mandate that all the control signals are acknowledged before initiating any communication. In [75], a syntactic profiling of the MPI application is performed to predict the relative arrival times of the MPI calls across all processes. Instead of static profiling, [51] profiles the application at run-

time over two iterations. Because of factors like operating system noise [17, 35, 10] and network sharing, there is no guarantee that the profiled results will stay consistent over multiple runs of the application. Therefore, predictive approaches are purely speculative. OS noise or jitter are subtle variations in the expected execution of an application caused due to the interaction of hardware, user-space software, kernel daemons and OS management operations. Also, in a shared resource facility, the profiled application may be run on nodes that share the network with other nodes running other applications. This may cause variations in message latencies due to congestion, even if the messages are of the same size and between the same peers. Both of these factors may introduce unexpected delays in the execution of MPI calls, causing the application to behave differently than the expected profile. Additionally, in run-time profiling [51], the profiling mechanism may itself attribute to variations in execution times.

The approach in [9] leverages triggered operations specified in **Portals 4** [63] to improve the rendezvous protocol. However, the triggered rendezvous protocol cannot deal with the MPI_ANY_SOURCE wildcard, so the approach falls back to the suboptimal default rendezvous protocol in such a scenario.

Hardware-Assisted Approaches

In the previous section, we discussed protocol improvement approaches to achieve communication/computation overlap. Overlap can also be achieved by effectively using features available in the network hardware. This section discusses such research proposals. For InfiniBand based networks, [43] presents a mechanism called TupleQ, in which each message matching tuple of rank, tag and context ID is mapped to a Shared Receive Queue (SRQ). If a send call is issued to an SRQ address that has not been posted yet at the receiver, then the HCA itself blocks the call until the SRQ gets posted. In other words, no blocking is required at the middleware, leading to a sender-side overlap. When the corresponding receive call is issued, the SRQ is created if it is not

already present. The SRQ for a particular tuple is created only once and its address is cached at the sender for future use. In this approach, there is a full overlap at both the sender and the receiver if the receiver has already arrived and the SRQ address is known to the sender before arriving. Another advantage of this approach is that it completely avoids the use of control messages. However, this approach suffers from two major disadvantages. First, this approach is not scalable. In complex applications running on large clusters, the number of SRQs can scale exponentially; and it is known that large scale creation of InfiniBand queues per MPI process is not scalable [44]. This issue is aggravated by the fact that the cached SRQs are not freed until the termination of the application. The second disadvantage is its inability to deal with wildcards like MPI_ANY_SOURCE and MPI_ANY_TAG. In such cases, this approach resorts to the inefficient traditional approaches that use software control signals.

Certain NICs provide support for the offloading of communication primitives. In [27, 28, 37] the authors investigate one such feature available on Mellanox HCAs called CORE-direct. Specifically, CORE-direct supports the offloading of collective communications. It is possible to offload rendezvous communications to CORE-direct as well. However, it requires the duplication of all queue pairs across all processes and this approach is not scalable [37].

Similarly, hardware progression threads were available on Quadrics' Elan and Myricom's Myrinet NICs. These are investigated in [62] and [84] respectively. At the time of writing, Bull eXascale Interconnect (BXI) is the latest interconnect to support offloading of communication primitives [20] and is based on the Portals 4 protocol [63, 72]. BXI NICs support the offloading of point-to-point communications, collectives, as well as one-sided communications. The limiting factors in these proposals are the cost and availability of the specialized hardware. Since these special features are provided by specific hardware manufacturers, they do not adhere to a standard like InfiniBand and require different API calls for their usage. This restricts the portability of the middleware.

Host-Based Approaches

In CPU-based *asynchronous message progression*, additional threads or processes are spawned along with the MPI processes and they are responsible for progressing the messages in parallel with the execution of the application threads. Each of these progress threads or processes may be assigned to an entire CPU core or be over-subscribed with others. As discussed in Chapter 2, polling and interrupts are the two approaches that can be used for message progression. Consequently, most asynchronous message progression proposals are based on them.

The polling based approach has long been regarded as the "silver bullet" [34] for asynchronous message progression and employed in popular MPI implementations like MPICH [55] and MVAPICH [56]. The idea involves polling for completions in a busy loop. To maintain the responsiveness, sleeping of the thread is avoided. So, this approach leads to a 100 percent CPU utilization. Its advantage is that it is more responsive than an interrupt based approach, but it leads to one of the two major disadvantages: non-optimal resource utilization and oversubscription. Since each MPI process is a distinct context, each of them needs to have its own progression thread as well. For single threaded MPI applications, this may lead to the occupation of half of the CPU cores by the progress threads, which may not be actively involved in progressing communications but nonetheless use valuable computing resources. Also, it is a common practice among MPI users to use most (if not all) of the available cores for application threads to maximize the parallelization of their jobs, but in this case, using polling threads would lead to oversubscription of the cores. Oversubscription is known to incur performance penalties, especially with a polling thread [34]. In a multithreaded application, both of these adverse effects are less severe but still very prominent.

The interrupt based approaches [46, 76] involve the use of a progress thread that sleeps until it is awoken. It is awoken by an interrupt that is caused due to the completion of a communication request. Upon waking, the thread progresses the recently arrived messages, requests another

interrupt and goes back to sleep again. The advantage of an interrupt based approach is that it does not consume any CPU cycles while it is sleeping. The disadvantage, however, is the overheads associated with it. They are: the interrupt cost, the cost of context switch that happens when the thread wakes up and the cost of locking into the progress engine. In [46], the cost of locking into the progress engine is replaced by an interrupt cost that is incurred in signaling the application thread to call the progress engine. In [76], the authors propose an interrupt-thread mechanism for the sender-initiated RDMA Read based rendezvous protocol (Figure 3.1). With this protocol, for non-blocking receive, there can be two scenarios, (a) the sender arrives before the receiver (Figure 3.1(a) and Figure 3.1(b)) or (b) the sender arrives after the receiver (Figure 3.1(c) and Figure 3.1(d)). In the latter scenario, an interrupt would be useful because the message can potentially be progressed earlier than the MPI_Wait at the receiver, leading to a receiver side overlap. In scenario (a) however, an interrupt will incur all the previously discussed overheads, without accomplishing anything useful. The proposal in [76] selectively generates interrupts and dynamically requests them to minimize the overheads but is not immune. For instance, consider a situation shown in Figure 3.4. Since MPI_Irecv1 does not find its RTS (from MPI_Isend1), the requests for interrupts will be turned on. Now, if the sender calls MPI_Isend for some other MPI_Irecv, say MPI_Isend2 for MPI_Irecv2, and if MPI_Irecv2 has not been posted yet then an undesired interrupt will occur, leading to all the previously discussed overheads.

[18, 19] and [87] describe schemes that *opportunistically* use CPU cores for message progression. In [18, 19] the communication can be offloaded to ltasks which are similar to tasklets available in the kernel-space. These are scheduled to run asynchronously on idle cores when they become available. If idle cores are not available then ltask execution is triggered by timers or at explicit polling points. Ltasks are essentially threads and the use of a progression thread requires thread safety mechanisms, the overheads of which are not inconsequential [82]. [87] presents

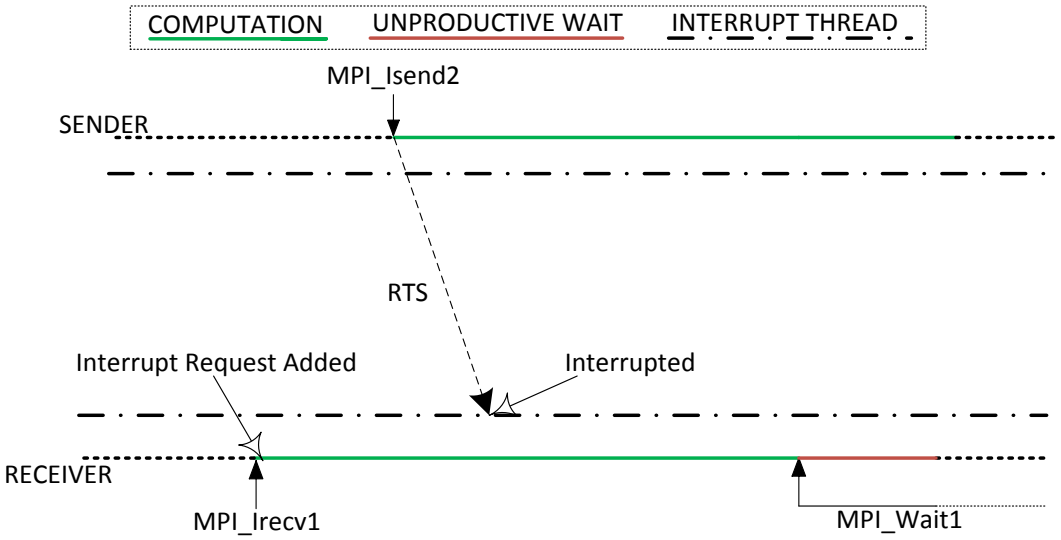


Fig. 3.4: Interrupt Thread based Asynchronous Progression

another opportunistic progression scheme which steals CPU cycles from the application thread to poll for message completions. As shown in Figure 3.5, this approach adds a parasitic execution flow to the default application execution flow. The application execution flow executes the program instructions and the parasitic execution flow executes the progress engine. By default, the parasitic execution flow is disabled and the CPU is entirely occupied by the application execution flow. However, if the application arrives at its MPI_Irecv and does not find its RTS, then the parasitic flow execution flow is enabled. From this point, until the message gets progressed, the application thread momentarily transfers the flow of execution from the application instructions to the parasitic instructions at regular intervals. If the RTS arrives while the parasitic execution flow is enabled, the message is progressed when the execution flow gets transferred to it.

[85] employs **MPI Profiling Interface (PMPI)** in its asynchronous progression proposal. In this paper, non-blocking point-to-point MPI calls and file I/O calls are redirected through PMPI to the progress thread. In the progress thread, MPI_Test or MPI_Wait may be called depending on the implementation. The problem with this approach is that it may lead to inheriting the shortcomings of polling or interrupts, depending upon the MPI implementation.

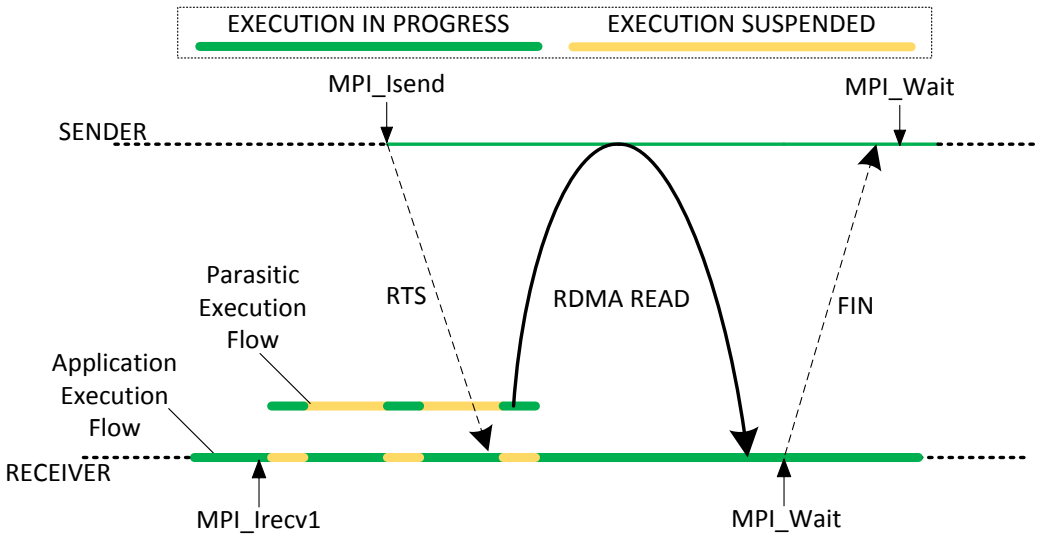


Fig. 3.5. Illustration of Parasitic Execution Flow Based Asynchronous Message Progression (Adapted from [87])

All the above mentioned host-based approaches can work well with co-processors/processors such as Intel Xeon Phi [69] as well. This is because of their similarity in architecture and supported instruction set with CPUs. For host-based overlap in GPUs, the authors of [31] propose a scheme to overlap GPU-to-GPU communications by over-decomposing the tasks into smaller sub-tasks and then over-subscribing the hardware with these sub-tasks with more threads than the hardware limits.

3.2 One-Sided Communication

3.2.1 Inefficiencies Associated with RMA Synchronizations

In RMA, a non-timely opening or closure of epochs may lead to unproductive waits at the peers. Unlike two-sided communication, the standard does not specify the blocking or non-blocking behaviour of the RMA synchronization calls. So the implementers are free to decide this nature as they see fit. Popular implementations of MPI [55, 56, 60] provide non-blocking epoch opening

calls for active target RMA synchronizations. The non-blocking nature of a synchronization call dictates the amount of communication/computation overlap that can be expected in its epoch. Also, it is important to note that most calls that issue RMA operations, such as MPI_Put and MPI_Get are inherently non-blocking when implemented using RDMA. The operation either completes immediately in the same call or gets deferred.

[8, 26, 78] try to circumvent the unproductive wait at the epoch opening by adopting a *lazy* approach, in which both the synchronization and the RMA operations are deferred to the epoch-closing call. To avoid the buffering of RMA operations, [81] suggests blocking the origin at epoch-opening until the target becomes ready for access. Similarly, to ensure timely execution of the RMA operations in a fence epoch, [71] suggests blocking at both epoch opening and epoch closure. This improves the overlap of RMA operations with computation but may cause a stalling of the early peers. All these approaches have one thing in common; they all block at the epoch closure. This leads to inefficiency patterns discussed in [33, 45, 88]. These inefficiency patters are as follows:

- *Late Post:* This happens in GATS when the origin has to block either at epoch opening (MPI_WIN_START) or epoch closure (MPI_WIN_COMPLETE) because the target has not opened its exposure epoch yet (MPI_WIN_POST). In the latest versions of MPICH, MVAPICH and OpenMPI, there is no blocking at epoch opening calls like MPI_WIN_START. This is important because a blocking behaviour at epoch opening has been shown to be suboptimal [8, 78].
- *Early Transfer:* This happens when an RMA operation call has to block because its exposure epoch has not been opened yet. The MPI 3.0 standard mandates a non-blocking behaviour for the RMA operation calls. Therefore, the early transfer inefficiency is expected to be absent in the latest implementations.
- *Early Wait:* In GATS, the target closes the exposure epoch by calling MPI_WIN_WAIT. However, this call blocks until all its origins have called MPI_WIN_COMPLETE. If

MPI_WIN_WAIT gets called before a target has finished all the RMA operations, then it would lead to an unproductive wait at the target.

- *Late Complete:* Similar to early wait, an unproductive wait is also inflicted upon the target when it has called MPI_WIN_WAIT in a timely fashion but the origin delays in calling MPI_WIN_COMPLETE after the last RMA communication call.
- *Early Fence:* This wait happens in a fence epoch when a peer calls the epoch closure function before all the RMA operations are complete. In fence, each peer is both a target and an origin, so neither of the peers are immune to this inefficiency.
- *Wait at Fence:* An epoch closing fence call blocks until all the peers associated with the window have issued the same function. Consequently, if a peer delays in closing its epoch after all its RMA operations, and if other peers have already called the epoch closure function, then that delay gets propagated to the rest of the peers as the Wait at Fence inefficiency.
- *Late Unlock:* Late Unlock is an inefficiency associated with passive target synchronization and can occur in both exclusive lock epochs and shared lock epochs.
    - In an exclusive lock epoch, this inefficiency can occur when the lock holding peer has completed all the RMA operations but delays in relinquishing the lock by calling MPI_WIN_UNLOCK. Meanwhile, if other peers are trying to acquire the lock then this delay gets propagated to them.
    - In a shared lock epoch, this happens when all the peers have completed their RMA operations but one of the peers delays in calling MPI_WIN_UNLOCK. Recall, that in a shared lock epoch, MPI_WIN_UNLOCK does not return until all the peers have called it.

3.2.2 Literature Review

This section discusses the research proposals that aim to improve the overlap by suggesting new non-blocking RMA synchronization calls or by exploring hardware-assisted or host-based approaches similar to Section 3.1.2. The above mentioned inefficiency patterns are addressed in [88], which proposes entirely non-blocking RMA synchronizations; even for epoch-closing routines. This is done by deferring the synchronizations to future RMA and progress engine calls, allowing overlap across multiple epochs.

Like two-sided communication, host based approaches have been proposed for RMA as well. These approaches include opportunistic message progression [88] and asynchronous message progression [39, 83, 73]. RMA GET/PUT operations do not need to involve the target as those calls are supported in hardware through RDMA Read/Write [40, 49]. However, for RMA operations involving non-contiguous data type like MPI_PACKED, the target has to be involved to unpack the data received on the contiguous temporary buffer and get the non-contiguous message. In [39], a thread based approach is used to asynchronously progress RMA operations involving non-contiguous data. The target also has to be involved in case of RMA Accumulate operations. Intra-node RMA communications require the active participation of the CPU as there can be no support from the NIC for such communications. In [86], the authors noted this inefficiency and also observed that in certain scenarios, inter-node RMA communications exhibit 100 percent overlap with some spare computation time. They exploit this residual overlap potential to overlap intra-node communications by deferring the transfer of such messages (above a threshold) to the spare computation time. [70] proposes a hardware-assisted approach to use InfiniBand's atomic functions like Compare-and-Swap and Fetch-and-Add to improve the overlap for RMA operations in passive target epochs.

Non-blocking routines are important to facilitate overlap but not enough to achieve ideal overlap. Similar to the Rendezvous protocol in two-sided communication, merely issuing a synchronization call is not enough to progress the communications. Certain conditions have to be met before the RMA communications can be initiated; otherwise, the requests are queued at the middleware and deferred to later calls. For instance, consider a situation in GATS where the origin has pending RMA operations and is involved in a long computation. Meanwhile, if the target opens its epoch then the pending RMA operations should ideally be initiated immediately. However, this cannot happen since the origin is busy in computation. An opportunistic message progression technique is proposed in [88] to address such inefficiencies. Similar to the two-sided version, this involves stealing CPU cycles from the application thread to progress pending RMA communications. [73] attempts to address these inefficiencies through an approach that uses process based asynchronous message progression with PMPI call redirection. These progression processes are called ghost processes, which can progress RMA communications for multiple MPI processes. Instead of calling the low-level network functions for RMA operations, the MPI processes use PMPI call redirection to offload this task to the ghost processes. The ghost process transparently redirects the call to the intended target and also progresses the RMA operations on behalf of the MPI processes that it is associated with. Since the ghost processes do not perform any computation, they are expected to be responsive in making RMA calls and progressing them. This approach uses shared memory so that the ghost processes can access the application data.

3.3 Summary

This chapter discusses the inefficiencies associated with point-to-point communication and RMA, and presents a literature review of the approaches that try to address them. Communication/computation overlap is the key parameter in deciding the efficiency of a messaging semantic, as serialization leads to suboptimal resource utilization which is undesirable in HPC.

In point-to-point communication, there is always a natural 100 percent overlap for eager messages because the application data is sent directly to the receiver, without any synchronization. In the rendezvous protocol, a natural overlap is possible only when certain conditions of process arrival patterns are met. For two-sided communication, the existing overlap proposals can be classified into three categories: protocol improvement, hardware-assisted approaches and host-based approaches. In an adaptive protocol improvement approach, the middleware automatically switches between receiver/sender initiated protocols based on the arrival order of the peers. Some adaptive approaches are augmented with buffering at the remote peer so that the local peer can be wait free in case the remote peer is very late. Other protocol improvement approaches involve profiling the communication pattern of an application and then choosing the protocols that can provide the maximum amount of communication/computation overlap. Profiling of the application can be performed statically through syntax analysis or dynamically during run-time. Hardware-assisted approaches rely on specialized features on the NIC that can be exploited to offload communication primitives or to provide asynchronous message progression. Finally, host-based approaches employ CPU or GPU cores to improve the overlap. The most common host-based approach is to use CPU cores for asynchronous message progression. In this approach, the asynchronous progress thread can be based on polling or interrupts. Other proposals include opportunistic message progression and PMPI call redirection.

In one-sided communication, serialization can happen due to a non-timely opening or closure of epochs. In GATS, this can happen if the exposure epoch is opened late or closed early, or if the access epoch is closed late. Serialization happens in Fence when one of the peers tries to close its epoch before all the RMA operations are complete or when a peer issues the epoch closing call significantly later than the rest of the peers. In Lock/Unlock (passive target) this can happen in an exclusive lock epoch if the lock holder delays in relinquishing the lock. Similarly, in a shared lock epoch, a delayed epoch closure by one of the peers can lead to the propagation of this delay to the

rest of the peers. One way to address these inefficiencies is by using entirely non-blocking epoch manipulating calls. RDMA can be used for high performance and asynchronous RMA operations; however, serialization can still occur on RDMA enabled networks in certain scenarios. For instance, the progress of RMA operations should ideally not require any intervention from the target but this cannot be avoided if the RMA operation involves non-contiguous data. This inefficiency was addressed using InfiniBand's atomic operations. Quite often, there may be scenarios where the target opens the exposure epoch while the origin is involved in a long computation. In such a scenario, the pending RMA operations cannot be progressed until the origin's computation is complete, leading to an unproductive wait at the origin and possibly at the target as well. This scenario can be handled through host-based approaches like opportunistic message progression and asynchronous message progression with PMPI call redirection.

From the discussion in this chapter, it can be inferred that almost all overlap approaches suffer from some kind of drawback. Also, there is no documented approach that can address the inefficiencies of both two-sided and one-sided communications. The next two chapters discuss the design and implementation of a novel asynchronous message progression technique that works well for both two-sided and one-sided communication. This approach addresses all of the two-sided inefficiencies and most of the one-sided inefficiencies while incurring negligible overheads.

Chapter 4

Node-Wide Asynchronous Message Progression for Point-to-Point Communication

Chapter 3 discusses the inefficiencies associated with the rendezvous protocol and surveys the literature on communication/computation overlap for two-sided MPI communications. With the sender initiated protocols, a natural overlap is only possible when the sender arrives ahead of the receiver in the RDMA Read based protocol. The RDMA Write based protocol uses an extra control signal and presents no scenario where a natural overlap is possible. On the other hand, the receiver initiated rendezvous protocols [61] do not comply well with some of the specifications in the MPI standards. Chapter 3 then discusses the different researches through which communication/ computation overlap can be achieved. One of the techniques discussed is a host-based approach called asynchronous message progression [34, 46]. This technique involves the usage of CPU/GPU/Co-Processor cores to spawn progression threads/processes that can asynchronously progress the communications initiated by the application threads. This chapter investigates the second research question posed in Section 1.2 by discussing the design, implementation and performance evaluation of a novel overlap approach based on asynchronous message progression. This approach, called **SmartInterrupts**, is a hybrid node-wide message progression technique

which utilizes the advantages of both polling and interrupts. In the rest of this chapter, the design and implementation of SmartInterrupts is followed by a discussion on the micro-benchmarks and application study used for its performance evaluation.

4.1 Motivation

Chapter 3 classifies the existing overlap approaches into three categories: protocol improvement methods, hardware-assisted approaches and host-based approaches. Protocol improvement approaches are mostly adaptive [66, 74, 75] or predictive in nature [51]. Adaptive approaches cannot efficiently deal with a scenario where both the peers arrive at almost the same time. In such a situation, these approaches have to resort to the suboptimal default protocol or impose expensive synchronizations. Some adaptive approaches [74, 75] are augmented with buffering at one of the peers but they pose a risk of buffer exhaustion. Predictive approaches assume that the communication pattern of an application can be profiled using static or dynamic techniques. However, such techniques cannot account for unpredictable elements like OS noise. SmartInterrupts is completely deterministic; hence it guarantees the correctness of the entire mechanism.

Hardware-assisted approaches exploit specialized features provided on the NIC to either offload the communications to the NIC [20, 27, 37] or asynchronously progress them [62, 84]. One deterrent to employing these approaches is the need for a specialized hardware. In modern multi-core and many-core architecture machines, it is not inconceivable to find a few spare cores that are not involved in any computation. These few spare cores are enough for SmartInterrupts to improve the application performance by improving the communication efficiency. Another problem with hardware-assisted approaches is that they inhibit code portability, which arises due to the fact that the specialized features are provided by specific vendors, and they may not adhere to an API standard like InfiniBand. SmartInterrupts' design is based on standard operating system concepts

and implemented using standard UNIX calls on top of InfiniBand. Also, the design is not dependent on InfiniBand and can work with any RDMA enabled interconnect that offers similar communication semantics.

Among host-based approaches, the most common overlap technique is asynchronous message progression [18, 19, 46, 76, 87]. In this technique, the progression thread can either be polling based or interrupt based. A polling thread is more responsive but it is resource intensive. This can lead to suboptimal resource utilization or oversubscription. Since each MPI process is a unique context with its own address space, each of them needs its own asynchronous progression thread. Therefore, for a single threaded application, this can lead to the wastage of half of the CPU cores or cause oversubscription on each CPU core. On the other hand, interrupt thread based approaches are resource efficient but as discussed in Section 3.1.2, they may lead to the generation of futile interrupts which incur several overheads. They are, the cost to generate an interrupt, the context-switch cost and either the cost of locking into the progress engine or the cost to signal the application to call the progress engine. SmartInterrupts is a hybrid asynchronous message progression technique that combines the responsiveness of polling and the resource efficiency of interrupts, while practically eliminating the detrimental effects of both the approaches. It utilizes a fraction of the CPU cores compared to polling and generates interrupts based on MPI's message matching tuple of rank, tag and context ID. Since the interrupt generation mechanism is completely deterministic and essentially based on MPI's message matching semantics, interrupts are only generated when they are useful.

4.2 Design of SmartInterrupts

To improve the communication/computation overlap of point-to-point communications, SmartInterrupts' design was based on asynchronous message progression. The motivation behind the design was to consume minimum amount of compute resources and avoid the issues associated

with protocol improvement and hardware-assisted approaches. However, asynchronous message progression has its own disadvantages which are discussed in depth in Chapter 3. As mentioned in Chapter 2, when an MPI communication completes, the network driver generates information about the completion of this communication. For instance, in case of the sender initiated RDMA Read based rendezvous protocol, this completion information is generated when the RTS arrives at the receiver. The progress engine then uses this information to progress the message by issuing the RDMA Read. The network API calls to get this completion information can be blocking or non-blocking. The interrupt based progress engine is designed using the blocking API call, which causes the calling thread at the receiver to go sleep until interrupted by the NIC when there is a work completion. On the other hand, the non-blocking API call must return immediately regardless of whether there was a communication completion or not. Therefore, a progress engine using the non-blocking API call must issue this call repeatedly until it finds the completion information of its MPI communication. This is the idea behind the polling based progress engine.

In asynchronous message progression, the main thread of each MPI process spawns a thread which keeps the progress engine active until the process terminates. This thread then progresses the communications when its application threads are busy in computations, ensuring the timely initiation of the communications and improving the communication/computation overlap. Since polling and interrupts are the only two options for the design of the progress engine, most of the existing asynchronous message progression techniques are also based on one of them. In interrupt based asynchronous message progression, the progression thread sleeps until it awoken due to a communication completion. The progression thread then takes the necessary actions and goes back to sleep again. This interrupt thread does not consume any CPU cycles while it is asleep but it is associated with several overheads that are described in Chapter 3. On the other hand, a polling based progression thread continuously polls for communication completions in a busy loop. When a communication completion happens, it takes the necessary steps and goes back to polling again.

The major advantage of the polling thread is its responsiveness. This is made possible due to the complete avoidance of kernel-userspace context switches by directly polling on the user-level data structures that are automatically populated by the NIC when the communications complete. However, polling on a busy loop incurs 100 percent CPU utilization, which ultimately leads to the wastage of compute resources as discussed in Chapter 3. Therefore, to harness the strengths of polling and interrupt based approaches and to avoid their shortcomings, the following design objectives were formulated for SmartInterrupts:

1) Hardware interrupts are expensive. At the minimum, each hardware interrupt requires the NIC to first generate a hard-interrupt on the Peripheral Component Interconnect Express (PCIe) bus. This hard-interrupt is then handled by the NIC's driver at the kernel and a soft-interrupt is triggered if a userspace thread is waiting for it. A large number of incoming messages may not result in soft-interrupts but lead to unnecessary hard-interrupts anyway. Therefore, a mechanism was required to minimize the hard-interrupts or eliminate it entirely if possible.
2) A soft-interrupt triggered to an expecting interrupt thread leads to a context-switch and a call to the progress engine. Also, the interrupt thread might have to contend for the progress engine lock. These activities are costly, but the costs are justified if the interrupt is useful. However, as mentioned earlier, this is not always the case. Therefore, a deterministic mechanism was required to make sure that the interrupts are only triggered when they are helpful.
3) The problem with the existing polling based approach is that each MPI process requires its own progression thread, which leads to the wastage of computing resources. This is because of the fact that the completion information data structure is only confined to the address space of individual MPI processes. However, the responsiveness of polling was desired in the design, which led to the following design objectives:

a) Associating multiple MPI processes with a single polling agent, so as to be resource efficient. This polling agent could be a tasklet at the kernel or a thread at the userspace.

b) Polling for the completion information of multiple MPI processes at the same place meant that the available network API function and its associated data structure could no longer be used to poll. Therefore, a mechanism was required to emulate the completion information data structure and share it across multiple contexts, which could be between the userspace process and the kernel or between multiple userspace processes.

These design objectives were accomplished by developing a hybrid approach that uses both polling and interrupts, but neither in the traditional sense. Like the interrupt based asynchronous message progression proposals, the progression thread in SmartInterrupts is an interrupt thread that progresses messages whenever interrupted. However, the interrupts are not generated by the NIC's device driver and not part of the network's software stack. Instead, the interrupts are triggered by polling processes called **Helper Processes (HPs)** that may be associated with multiple MPI processes. This addresses Objective 1 by completely eliminating hardware interrupts and also addresses Objective 3(a). To achieve Objective 3(b), a shared buffer was created between each MPI process and its associated Helper Process. This shared buffer contains information about the incoming control messages related to the rendezvous protocol. In SmartInterrupts' design, the data in this buffer emulates the completion information. Finally, to achieve Objective 2, the interrupt generation mechanism was based on MPI's message matching semantics. This guarantees that an interrupt will only be generated when it is helpful. As discussed in Chapter 2, MPI's message matching is performed on two groups of data, the messages that have arrived at an MPI process and the communication requests that are queued locally. Information about the former was shared with the helper processes when Objective 3(b) was addressed. Information about the latter was

shared by creating another shared buffer between the MPI process and its Helper Process. The Helper Process matches the contents of these two buffers and triggers an interrupt to the interrupt thread upon finding a match. This causes the sleeping interrupt thread to wake up and then asynchronously progress the message while its application thread is busy in a computation.

4.2.1 Choosing the Default Rendezvous Protocol

Chapter 3 discusses the sender and receiver initiated rendezvous protocols, along with their RDMA Read and Write based implementations. The receiver initiated protocols suffer from the inability of dealing with certain specifications of the MPI standard. The sender initiated RDMA Write based protocol uses an extra control signal and presents no scenario of arrival orders where the communication can naturally overlap with the computation. On the other hand, in the sender initiated RDMA Read based protocol, an overlap can be expected if the sender arrives ahead of the receiver. However, if the sender arrives later than the receiver then the communication gets deferred to the MPI_Wait. This means that an external message progression support would only be required in the latter scenario. SmartInterrupts is designed around the sender initiated RDMA Read based rendezvous protocol, so that it can leverage its natural overlap potential. In scenarios where there is a possibility of a natural overlap, SmartInterrupts will not have any detrimental impact on the performance.

4.2.2 Asynchronous Message Progression Mechanism

Figure 4.1 illustrates the data-movement and signaling involved in SmartInterrupts. When the sender arrives at MPI_Isend (C), it sends a modified RTS control message to the receiver. This modified control message has two parts. The first part is the original unmodified RTS control message that contains the message envelope and the second part is a duplicate of the message matching information of the first part. When this modified RTS control message arrives at the

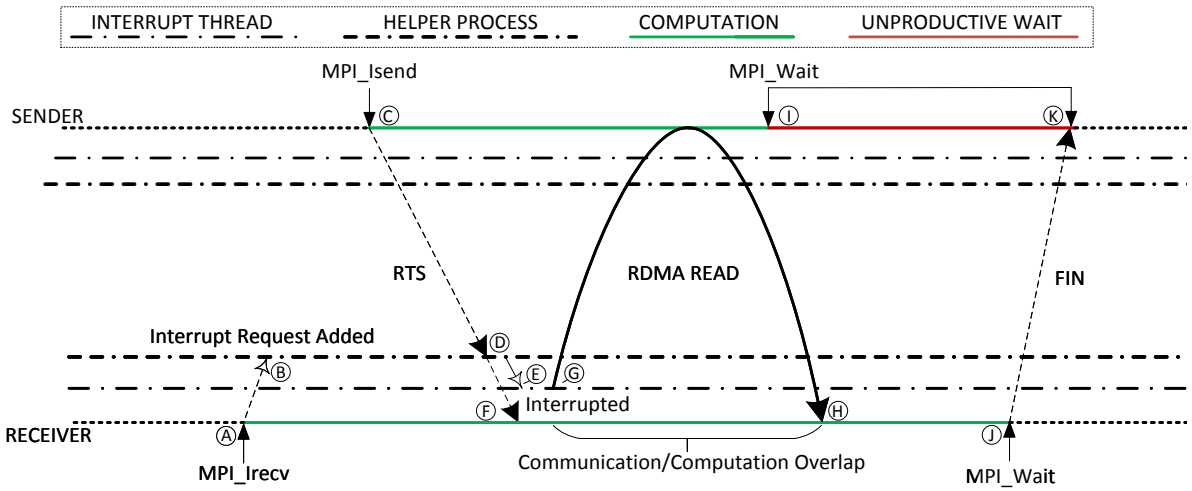


Fig. 4.1. Data Movement and Signals in Smart Interrupts

receiver, the first part gets copied to the receiver's communication buffer (F) and the second part gets copied to the shared buffer between the receiver and its helper process (D). According to Objective 3(b), this is required to make the helper process aware of the incoming RTS. When the receiver arrives at the matching MPI_Irecv call (A) and does not find its RTS, then it requests for an interrupt from its helper process (B). It does so by adding a request to the other shared buffer that was added to accomplish Objective 2. The helper processes continuously poll on the incoming RTSs since spawning and try to match them with the interrupt requests submitted to them (B). An interrupt is triggered to the receiver's progression thread if a match is found for its request. In the illustrated scenario, the MPI_Irecv's request is already present in the shared buffer when the RTS arrives at the receiver. The helper process eventually finds the matching RTS for the initially submitted request and triggers an interrupt to the interrupt thread (E). This causes the interrupt thread to wake up and call the progress engine. The progress engine then progresses the message by issuing the RMDA Read (G). Since the communications are offloaded to the NIC, the interrupt thread can immediately go back to sleep after issuing the communication calls. For the illustrated scenario, this leads to a communication/computation overlap between (G) and (H). Since the communication is entirely progressed before the call to the MPI_Wait at (J), the receiver does not

have to block and can send the FIN control signal instantly, which is also offloaded. On the other side, the delay at the sender's MPI_Wait depends upon when it gets called, as it cannot return until it receives the FIN control signal. In this case, the sender has to wait between (I) and (K), however, it could have been wait free if it had issued the call after the arrival of FIN (K). There are three important optimizations in the design to make the helper processes more efficient and to eliminate any possibility of misfiring interrupts. The details of these optimizations are discussed in Section 4.3.

4.2.3 Core Components

Figure 4.2 illustrates the important components of SmartInterrupts, in which each MPI process has one application thread and one interrupt thread for asynchronous message progression. The application thread is the main thread which can be safely oversubscribed with the interrupt thread, as the latter does not poll periodically and lays dormant until interrupted. While the thread sleeping, it is taken off the scheduler's run queue and it does not consume any CPU cycles in this state. Other important entities are the **Interrupt Handler** kernel module**,** the Helper Processes and the two shared memory regions between the MPI processes and the helper processes.

As mentioned earlier, the task of interrupt generation is the responsibility of the helper processes. As the name suggests, they assist the interrupt threads and in turn assist the MPI processes to asynchronously progress messages. However, it is important to note that they only trigger the interrupts. They do not call any network related API functions, and calling the progress engine is still the responsibility of the asynchronous progression interrupt thread. The interrupts are triggered based on the data in the shared buffers, which bridges the MPI process to its helper process. The number of helper processes per node can be specified as a command line argument at the time of execution. However, it is important to note that a minimum of one per node is required

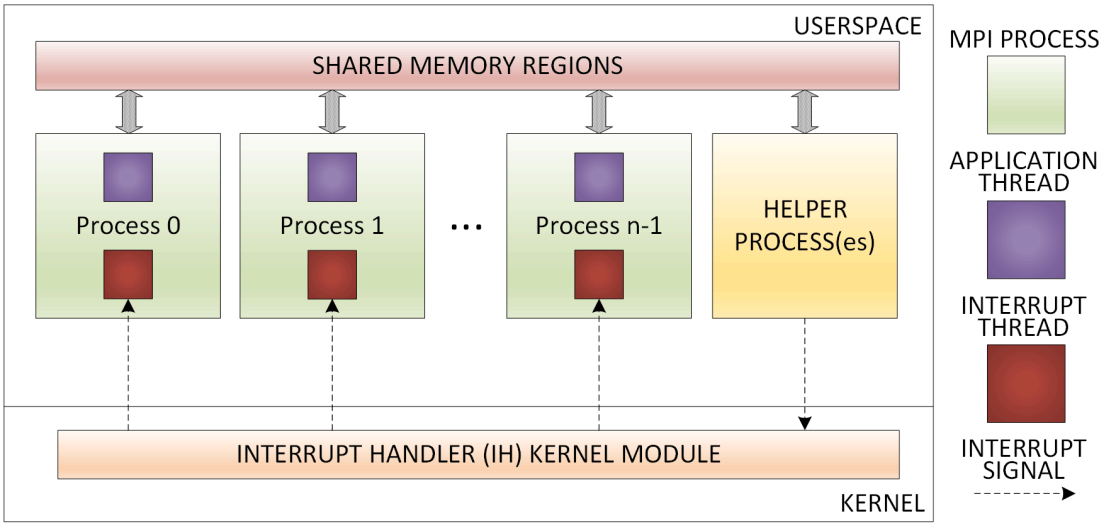


Fig. 4.2 Core Components of SmartInterrupts

to trigger any interrupt to the progression threads. The helper processes use a busy loop, so for maximum performance, each of them needs to run on an individual CPU core. The objective of this approach is to use only a small percentage of CPU cores and still provide a decent amount of overlap. In modern multi-core and many-core architectures, dedicating just a few cores for asynchronous message progression is more beneficial than using all the cores for MPI processes with no asynchronous progression at all.

4.3 Implementation of SmartInterrupts

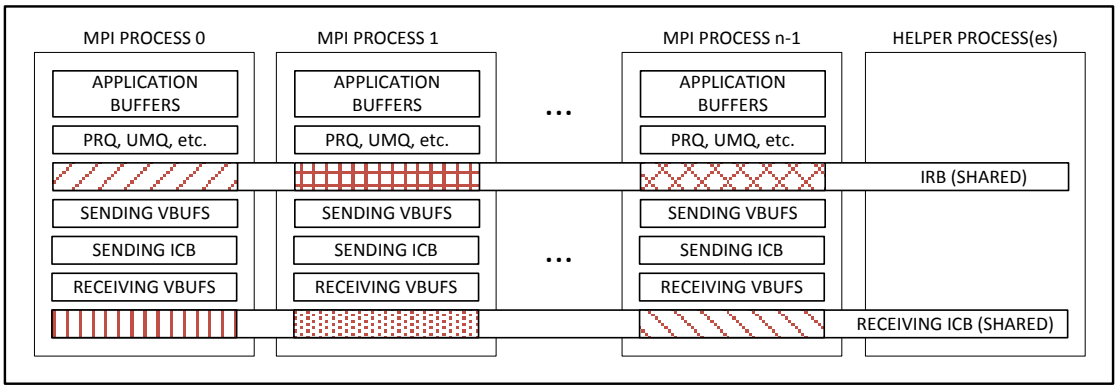
4.3.1 Interrupt Handler Kernel Module

The Interrupt Handler kernel module assists in accomplishing the Design Objective 1 by completely eliminating the need for hardware interrupts. It does this by creating a virtual file in the **proc virtual filesystem** and defining the **read()** and **write()** functions associated with this file. This module creates an array of wait queue elements, one for each MPI process. A wait queue is a list of processes, all waiting for a specific event. However, in this kernel module, each wait queue is associated with only one process. The state of the interrupt threads is manipulated in the read() and

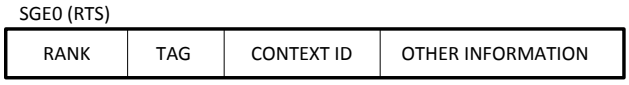
write() functions by controlling the wait queues. To go to sleep, an interrupt thread must issue the read() system call on this virtual file. Similarly, generating an interrupt to a specific sleeping thread requires a call to the write() system call on the virtual file by the helper process. To ensure the sleeping and waking of the correct interrupt thread, the local MPI rank is passed as a parameter to these functions.

4.3.2 Shared Buffers

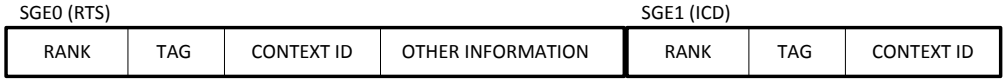
SmartInterrupts is implemented on MVAPICH2 [56], which supports InfiniBand [36] through OFED [59]. Necessary modifications were made to the buffer structures of MVAPICH2 to address the different design objectives of SmartInterrupts. Figure 4.3(a) illustrates the different buffers that are present in the implementation of SmartInterrupts. In MVAPICH2, the registered memory regions used for Send/Receive InfiniBand communications are called **Virtual Buffers (VBUF)**, and they are of two types, sending VBUFs and receiving VBUFs. To the existing pool of VBUFs, one more is added to each of the types. These are called sending and receiving **Interrupt Control Buffers (ICB)**. The receiving ICB is part of a shared memory region between the MPI process and its Helper Process. This is the same shared buffer that was used to address the Design Objective 3(b) and is one of the components that was required for the Design Objective 2. In SmartInterrupts' design, each MPI process has a distinct buffer shared with its helper process. However, in the implementation, rather than creating separate shared buffers between each MPI process and its helper process, a single shared memory region is created among all MPI processes and the helper processes, as shown in Figure 4.3(a). However, this shared buffer region is strided, with each stride allocated to an MPI process for its shared space with its helper process. Therefore, although ICB and IRB may contain information about multiple MPI processes, there is no communication between the MPI processes through these shared buffers.



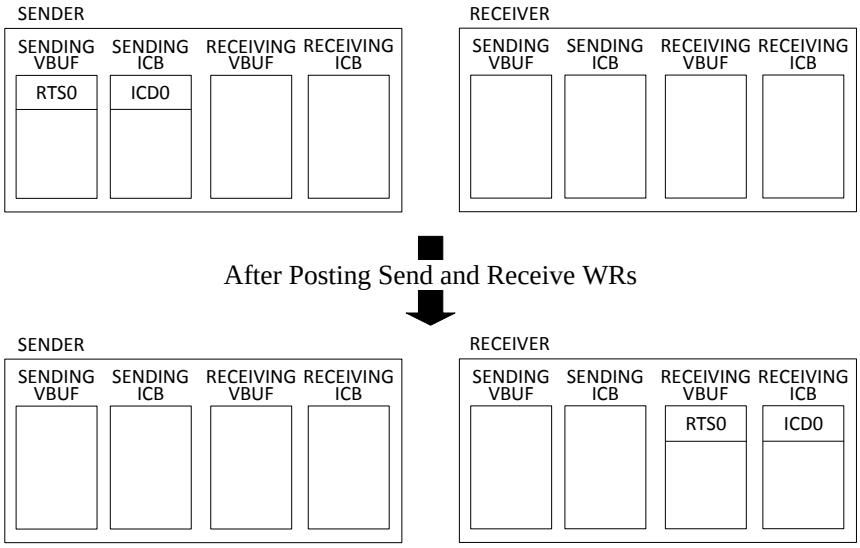
(a) Organization of Buffers



(b) Unmodified RTS from vanilla MVAPICH



(c) Modified RTS in SmartInterrupts



(d) Transfer of RTS from Sender to Receiver

Fig. 4.3. Illustration of Buffers and Data Movement in SmartInterrupts

As stated before, control messages like RTS are sent eagerly. MVAPICH2 provides two ways of sending eager messages, Send/Receive and RDMA Write (Fast Path [42]), but the former suits

the design better as it allows storing data in a non-contiguous remote buffer. In vanilla MVAPICH2, the eager message is copied to the sending VBUF and a send WR is created with this VBUF as the Scatter Gather Element (SGE). Recall, that an SGE points to a memory region on which DMA operations like read and write can be carried out (Section 2.4.1). In InfiniBand's Send/Receive semantics, while processing a send WR, the NIC reads from this SGE and transfers the data to the destination. In order for this data to be successfully stored somewhere at the destination, a matching receive WR must be posted at the destination. This receive WR must specify one or more SGEs on which the incoming data may be stored. When the send WR is posted, the data from the sender gets ultimately stored at the SGE(s) that are specified in the receive WR. Therefore, at the receiver in vanilla MVAPICH2, several receive WRs are posted in advance, with each WR containing the address of a single receiving VBUF as the SGE. When the send WR is posted, the eager message ends up at one of the receiving VBUFs at the receiver. In SmartInterrupts' implementation, these operations as modified as follows:

a) At the sender, the eager message is copied to the sending VBUF like before. In addition to that, if the eager message is an RTS then the message matching tuple (rank, tag and context ID) associated with that message is copied to the sending ICB (Figure 4.3(c)). This message matching tuple is referred to as the **Interrupt Control Data (ICD)**. If the eager message is not an RTS then it is appropriately indicated in the ICD. Now, in each send WR, two SGEs are specified, the first containing the address of the ICD in the sending ICB and the second containing the address of the message in sending VBUF.

b) Similarly, at the receiver, each receive WR is posted with two SGEs instead of one. The first SGE points to a memory location in the receive ICB and the second points to one of the receiving VBUFs.

When the send WR is posted, the eager message lands at one of the receiving VBUFs and the ICD at the receiving ICB (Figure 4.3(d)). This receiving ICB is shared with one of the helper processes.

Consequently, the ICD becomes available to the receiver's helper process at point (D) in Figure 4.1.

In addition to the sending and receiving ICBs, another shared buffer, called the **Interrupt Request Buffer (IRB)** is added between the MPI processes and their helper processes, but this buffer is not registered with the HCA. The IRB and the ICB together form the two components required for the Design Objective 2. If the receiver arrives at MPI_Irecv and does not find its RTS in the Unexpected Message Queue (Section 2.3.1) then it adds a request to the Posted Receive Queue, as well as an entry to the IRB at point (B) in Figure 4.1. This entry, called the **Interrupt Request Data (IRD)** is essentially a request to its helper process to trigger an interrupt to its interrupt thread. As stated earlier, although ICB and IRB may contain information about multiple MPI processes, there is no communication between the MPI processes through these shared buffers.

4.3.3 Helper Process and Interrupt Thread

At this juncture, the helper process knows about the RTSs that have arrived, as well as the receive requests that are looking for them. Therefore, it has enough information to trigger an interrupt and progress the message asynchronously if a match is found between the ICDs and IRDs. This sequence of actions is illustrated in Figure 4.1. However, this design is augmented by the following optimizations which improve the efficiency of message matching, eliminate the triggering of futile interrupts and minimize inter-processor communications.

Optimization 1: Improving the Efficiency of Message Matching in Helper Processes

The helper processes continuously poll on their ICBs to look for RTSs. If the progress engine is already active in an MPI process then it will most likely progress the pending messages of that process. In such a case, polling for its ICDs at the same time by the helper process serves no purpose, leads to wastage of CPU cycles and delays the message matching of other MPI processes that are associated with the helper process. Therefore, in addition to storing the IRDs, a region in

the IRB also contains information about the progress engine semaphore. If the progress engine lock is held by the main thread, then that information is available to the process's helper process as well. This information enables the helper processes to avoid performing message matching for the MPI processes in which the progress engine is active and focus on other MPI processes whose lock has not been acquired. This is one of the three optimizations that was mentioned in Section 4.2.2. This speeds up the message matching process for the MPI processes where the progress engine is not active.

Optimization 2: Eliminating Futile Interrupts

Upon finding an RTS, message matching is performed on the IRDs. If a match is found, then an interrupt may be triggered to the interrupt thread of the appropriate MPI process. However, it is possible that the MPI process acquires the progress engine lock during the message matching process. As mentioned earlier, if the progress engine is already active in the main thread, then it will most likely progress the message too. Generating an interrupt in this case would not only be futile but also incur several unnecessary overheads. To avoid this, the status of the MPI process' progress engine lock is examined before triggering the interrupt, and the interrupt thread is only awakened if the lock has not been already acquired. This information about the lock was made available through Optimization 1. This second optimization is an important aspect of the design as the lack of it can significantly hamper the performance. There is a possible scenario in which the progress engine in the main thread might be almost at the end of its call, with no chance of the message being progressed by the main thread in that call. From the helper process's perspective, the progress engine's lock would be acquired, so it would not trigger an interrupt immediately. In this situation, the opportunity for interrupt generation is not lost but slightly delayed.

Optimization 3: Minimizing Latency due to Inter-Processor Communications

Due to the inter-process communication between the MPI processes and helper processes, careful consideration is required about the locality of the shared buffers and the relative position of an MPI process with respect to its helper process. As mentioned in Chapter 2, modern compute nodes may have several NUMA nodes upon which the processes are running. A NUMA node consists of a CPU and one or more memory modules that are local to it. Data transfers across NUMA nodes are facilitated by the inter-socket interconnect, and such transfers are slower than accessing the local memory. In fact, in modern processors, the CPU cores may share their caches with each other which further decreases data access latencies. Therefore, to maximize the performance of SmartInterrupts, the MPI processes and their helper processes are mapped to the CPU cores of the same NUMA node if possible. This ensures that message matching information is available to the helper processes as soon as they arrive, so that there is minimum delay in the generation of interrupts. This mapping of the processes to the CPU cores is done with the help of a software package called **Portable Hardware Locality** or **hwloc** [13].

Performance Improvement in Other Scenarios

The previous sections emphasize on the usage of SmartInterrupts in the scenario where the receiver arrives ahead of the sender. Regardless of the MPI implementation, the proposed approach would be useful in this scenario. But depending on the implementation and the application, SmartInterrupts may also be useful in the other scenario where the sender arrives first. As discussed in Section 2.3.1, an eager message like RTS does not land directly into the UMQ but instead into an intermediate communication buffer. It is the job of the progress engine to move this stray RTS from the intermediate buffer to the UMQ if the receiver has not issued the matching receive call yet. In certain MPI implementations, an MPI_Irecv call internally calls the progress engine first. In such a case, there will be a natural communication/computation overlap and SmartInterrupts'

support will not be required. Its support will also not be required if the RTS has already arrived and the receiver explicitly calls the progress engine before issuing the MPI_Irecv call. However, if there is no implicit or explicit call to the progress engine then the proposed approach would be able to improve the overlap, even in the scenario where the sender arrives first.

With the proposed design and the added optimizations, there is no possibility of incorrectly triggering interrupts. Also, it does not lead to the wastage of half of the CPU cores. Consequently, it addresses most of the shortcomings of the polling and interrupt based approaches discussed in the Section 3.1.2. But there are still some unavoidable overheads which are inherent to an interrupt based system; like, the cost to generate an interrupt and the cost of a context-switch. Also, the interrupt thread gets oversubscribed with the main thread when it awakens. However, this oversubscription lasts for a very short duration, as the interrupt thread has to only post the WRs for RDMA Reads and not wait until the transfers are over. One might also be concerned about the overhead added by the extra information (ICD) that is sent along with each eager message, but the experimental evaluation shows that it is negligible. The details are discussed in Section 4.5.

4.4 Design Alternatives

Design Alternative 1: Using Hardware Interrupts

Alternative design approaches were considered but were not followed through due to some inherent drawbacks. In one such approach, there was no concept of a helper process and it did not involve polling at the userspace. The idea behind this approach was to use InfiniBand's event based completion notification and to intercept the HCA's interrupts at the kernel. This approach also relied on the usage of shared buffers, but between the HCA's device drivers and the MPI processes. Information about the interrupt requests and incoming control messages would be made available to the drivers through these buffers. Also, this approach required the network APIs to be used in a

way such that a hardware interrupt is generated whenever a control message arrives at the receiver. A hardware interrupt activates the interrupt handling part of the device drivers. That code would then examine the contents of the shared buffers and match the interrupt requests to the control messages that have recently arrived. The interrupt would be allowed to pass to the userspace in case of a match, otherwise, kept buried at the kernel.

It is evident from the description that this approach would have required a modification of the HCA's device drivers. Supercomputing facilities are generally made available through agencies that promote research, however, access to these machines is very limited and it is very unlikely to get a permission for driver modification, as it may cause adverse affects on the applications of other users. Despite this limitation, this idea was explored for some time but it resulted in the discovery of two more issues, soft-lockups and race conditions. Long computations at the kernel can lead to soft-lockups which inhibit the timely execution of other important operating system activities, rendering the machine unstable. An interrupt thread based on pthreads cannot be terminated by its process if it is asleep. Also, in this approach, interrupts can only be triggered through a communication from another node. So, for a successful termination of the application, it is imperative that all interrupt receiving blocking calls are matched by an appropriate communication. This is easy to handle if each MPI process has exactly one application thread. However, if there are multiple threads per process, each having the ability to issue the blocking interrupt call, then only one of the threads will be awakened if several of them issue the blocking call together, causing a race condition where a few of the threads may block forever.

Design Alternative 2: Using the Immediate Data Field in the WR Data Structure

The approach proposed in this thesis adds extra information to each eager message. This information consists of the message matching tuple of rank, tag and context ID, which is contained in an SGE of eight bytes. The next section will show that this addition has no effect on the performance, however, an approach was considered to avoid this. In InfiniBand, there is a special

field in the work request's data structure called **Immediate Data (imm_data)**. It is different from the RDMA payload because it does not get stored into the remote application buffer, but instead gets stored as work completion data. If it is confined to the size limit, this field can be set to any arbitrary value and can be sent along with the payload. However, the size of this field is limited to 32 bits in hardware, which is not sufficient to store the entire message matching tuple, which by default is 64 bits and can be even larger for large clusters.

4.5 Performance Evaluation and Analysis

This section evaluates the performance, overheads and scalability of SmartInterrupts for point-to-point communication. To thoroughly evaluate the design, the implementation was used as a middleware to test several two-sided micro-benchmarks, one collective micro-benchmark and one scientific application. The same micro-benchmarks and application were then executed on other available solutions, and the results were analyzed.

4.5.1 Description of Hardware and Software

The cluster used to execute the micro-benchmarks and application consists of 32 nodes, all connected to a single Switch-IB SB7700 switch. Each node is equipped with two 10-core Intel Xeon CPUs (E5-2680) running at 2.8GHz, 64GB DDR3 memory and a Mellanox [52] ConnextX-4 HCA. The software environment consisted of the Red Hat Enterprise Linux 7 OS with kernel version 2.6.32-431, Mellanox OFED 3.4-1 and MVAPICH2-2.2a.

4.5.2 Two-Sided Micro-Benchmarks

The performance of point-to-point micro-benchmarks was evaluated using two nodes of the cluster described above. The evaluation was based on the parameters of latency overhead, communication/computation overlap, asynchronous message progression, scalability and memory

footprint. The design of the micro-benchmarks was inspired by the ideas suggested in [65]. The evaluation of the proposed design requires several pairs of senders and receivers communicating in parallel. This is because SmartInterrupts is a node-wide asynchronous message progression proposal and its performance can be best evaluated when each helper process is assisting multiple MPI processes on the same node. One of the nodes is designated as the sending node and the other as the receiving node. Each MPI process (MP) in the sending node communicates exclusively with one MPI process in the receiving node. This pair-wise communication scheme is illustrated in Figure 4.4. Before the start of each communication, the senders and receivers are synchronized so that all the MPI_Isends are posted together and all the MPI_Irecvs are posted together. Also, since the design is aimed at receiver side overlap, all the timing measurements are performed at the processes in the receiving node.

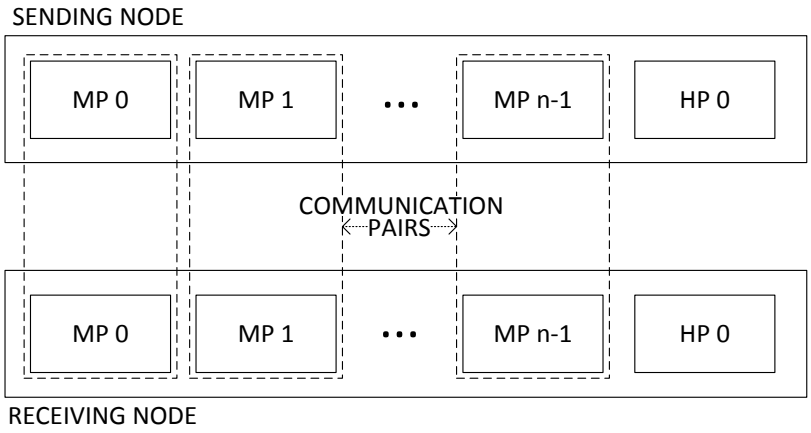


Fig. 4.4. Pair-wise Communication in Two-Sided Micro-Benchmarks

Figure 4.5 shows the generalized sender and receiver algorithms that were used to analyze the overhead, overlap, progression and scalability. In all the point-to-point micro-benchmarks, 10,000 iterations of these algorithms were used and the first 200 iterations were discarded to account for cache warm-up. In both the algorithms, the sender and receiver start each iteration with a call to MPI_Barrier. This ensures that the following lines are executed at almost the same time. Line 2 at the sender is there to simulate the scenario where the receiver arrives ahead of the sender and misses

SENDER:	RECEIVER:
1. MPI_Barrier(MPI_COMM_WORLD) 2. Small constant delay to assure that the receiver arrives first 3. MPI_Isend to the appropriate receiver 4. MPI_Wait	1. MPI_Barrier(MPI_COMM_WORLD) 2. Measure Start Time 3. MPI_Irecv from the appropriate sender 4. Variable synthetic work according to the micro-benchmark 5. MPI_Wait 6. Measure Stop Time

Fig. 4.5. Template for Two-Sided Micro-Benchmark Design

its RTS. This is the scenario where SmartInterrupts would be most helpful. However, this delay does not exist in the latency overhead micro-benchmark. The reason behind this can be found in the next section. After the delay, the sender issues the MPI_Isend call and blocks at MPI_Wait until it receives the FIN control signal from the receiver. At the receiver, the timing measurement starts after the MPI_Barrier. Then, the MPI_Irecv is issued followed by a simulated computation at Line 4. The major difference between the individual micro-benchmarks is the implementation of this Line 4 in the receiver's pseudo-code and the processing of the results. Finally, the MPI_Wait for the MPI_Irecv is issued and the timing measurement is stopped.

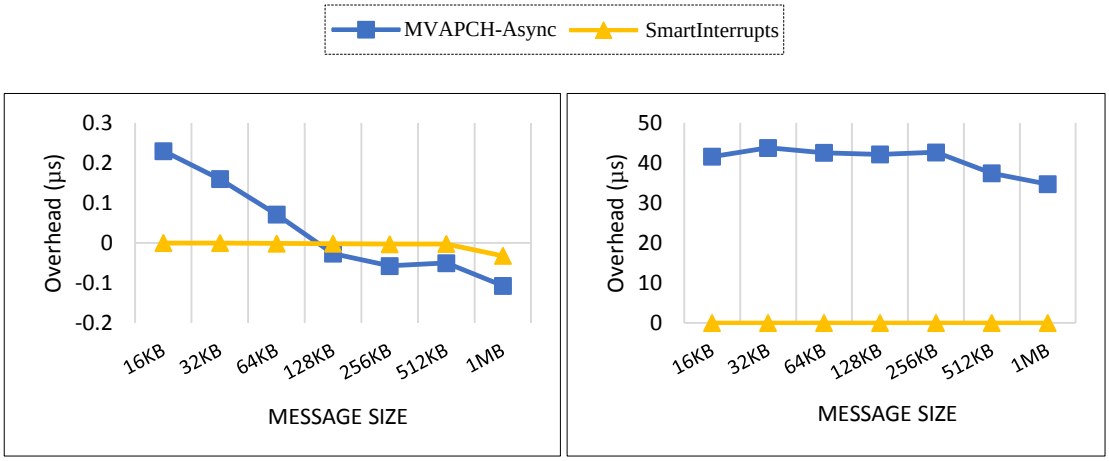
The results of SmartInterrupts are compared to that of MVAPICH2's polling based asynchronous progression and with its default, no asynchronous progression setting. These are respectively referred to in this chapter as MVAPICH-Async and MVAPICH. The results are not compared with OpenMPI because at the time of writing, it had no support for asynchronous progression. On the machines that were used, the default eager threshold of MVAPICH2 is 16KB and all of the micro-benchmarks were executed with messages sizes of 16KB-1MB. The micro-benchmarks were experimented with several different configurations of MPI processes and helper processes on each node, however, this chapter will limit the results to two configurations for the ease of comparing between the significant results. The first with 9 MPI processes and 1 helper process per node (9-1) and the second with 18 MPI processes and 2 helper processes per node (18-2). This is to be able to compare the performance of SmartInterrupts with both non-oversubscribed

and oversubscribed scenarios of MVAPICH-Async. Note, that the interrupt threads in SmartInterrupts remain oversubscribed with the main thread even if there are spare cores available. However, as discussed in Section 4.4, the overhead due to this mapping scheme is negligible.

Latency Overhead

The latency overhead ($T_{overhead}$) is the difference between the message latencies (T_{lat}) of the asynchronous message progression approaches and MVAPICH. In this micro-benchmark, there is no delay between the MPI_Irecv and MPI_Wait (Figure 4.5). Also, Line 2 at the sender is absent, so the sender and receiver arrive at almost the same time. Triggering interrupts would serve no purpose if the MPI_Irecv is immediately followed by the MPI_Wait. Moreover, it would be detrimental to the performance due to the overheads associated with interrupts. These algorithms help to analyze such overheads by simulating the scenario where SmartInterrupts would not be helpful. The averages of the latency values across all the processes is calculated and used for the overhead measurement.

Figure 4.6 compares the overhead of Smart Interrupts with MVAPICH-Async for different message sizes. In Figure 4.6(a), the overheads of both implementations are negligible, in the order of nanoseconds. The overhead curve for MVAPICH-Async starts leaning to the negative side as the message size increases. This is because of OS noise and network contention. The execution times of MPI_Isend and MPI_Irecv calls, and the message latency of control signals are comparable. Also, since all the MPI processes issue their MPI_Isend at almost the same time, the network contention can cause slight perturbations on the arrival of RTSs. Therefore, it is possible that an RTS may not arrive at its receiver when the MPI_Irecv call is issued. If it does not, then the progression gets deferred to the next call to the progress engine. In MVAPICH, this can only happen in the next MPI_Wait call, and there is some latency between the missing of the RTS and the next



(a) 9 MPI processes per node, 1HP for SmartInterrupts (b) 18 MPI processes per node, 2HPs for Smart Interrupts

Fig. 4.6. Two-Sided Latency Overhead Results over MVAPICH

call to the progress engine. This latency is recovered with MVAPICH-Async as the progress engine is always active; therefore, the message can be progressed as soon as the RTS arrives.

In Figure 4.6(b), SmartInterrupts continues this trend; however, the overhead of MVAPICH-Async significantly increases. This is because, there are 18 MPI processes on each node which causes the oversubscription of main threads and poling threads. In SmartInterrupts, oversubscription only happens when the interrupt thread is called into action, which is unlikely in this micro-benchmark because of the immediate arrival of MPI_Wait after MPI_Irecv. From the results, it can be concluded that SmartInterrupts incurs negligible overheads and does not adversely impact the performance.

Communication/Computation Overlap

For the measurement of communication/computation overlap, the previously mentioned algorithm (Figure 4.5) was modified according to [65]. In the first iteration, the synthetic work (T_{syn}) across all the receiving processes is kept zero and the message latency (T_{lat}) is calculated. This latency becomes the basis for the synthetic work in the following iterations. Then, for each

iteration, the synthetic work is increased by 10 percent of T_{lat} until it becomes 1.1 times of T_{lat}, that is:

$$0.1T_{lat} \leq T_{syn} \leq 1.1T_{lat}.$$

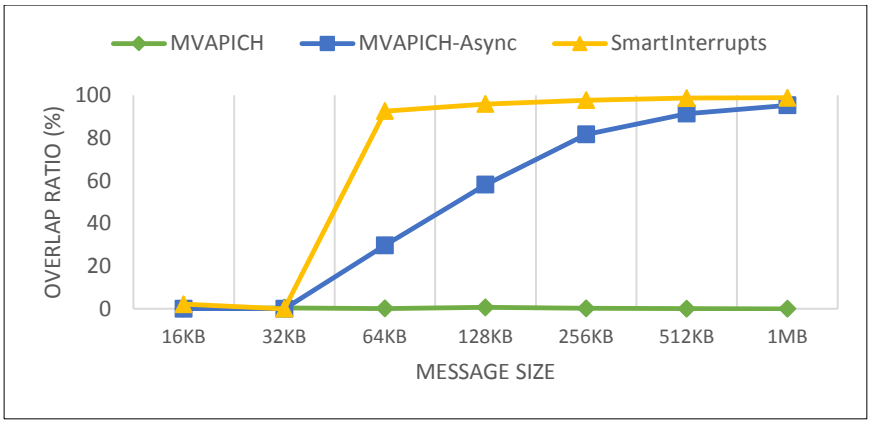
These values are based on empirical observations. To measure the overlap of a particular communication on a scale of 0 to 100 percent, the computation, which is derived from the communication's latency, is also varied at the same scale. For all values of T_{syn}, the elapsed time (T_{et}) is calculated by each of the receiving processes, along with T_{diff} and overlap ratio, where:

$$T_{diff} = 1.1T_{lat} - T_{et} \quad (4.1)$$

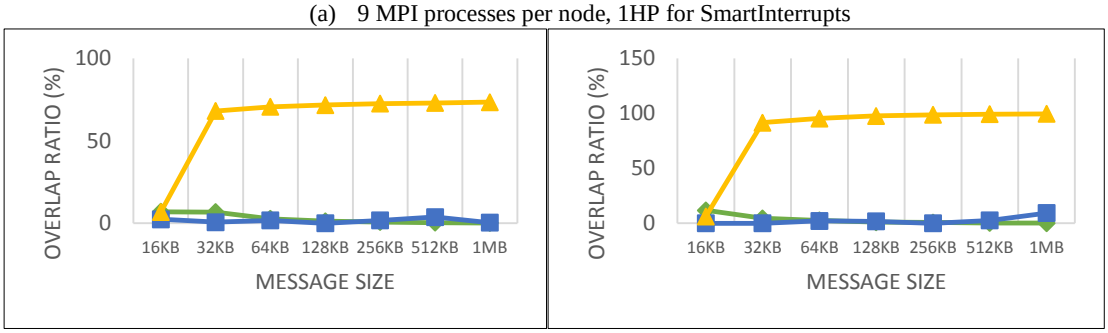
$$Overlap\_ratio = \frac{T_{syn} - (T_{et} - T_{lat})}{T_{lat}} \quad (4.2)$$

When the iterations are over, all the receiving processes send this information to the local process zero of the receiving node. The largest iteration for which T_{diff} is either positive or zero across all processes is used for the calculation of average overlap ratio.

Figure 4.7 shows the communication/computation overlap for 9, 18 and 17 MPI processes. As expected, MVAPICH exhibits negligible overlap in all scenarios. For 9 processes, both the asynchronous progression techniques exhibit appreciable overlap but SmartInterrupts performs better than MVAPICH-Async, even though the polling threads were not oversubscribed. SmartInterrupts achieves close to 100 percent overlap for messages greater than 64KB. For messages of sizes 16KB and 32KB, the message latencies are very small, leading to even shorter synthetic work. Also, the computation required for their processing is comparable to their communication time. These reasons reduce the overlap potential and explain why neither of the implementations show any overlap for those messages. For 18 processes, the results of MVAPICH-Async suffer due to oversubscription but SmartInterrupts shows overlap in the range of 70-75 percent for messages greater than 32KB. However, this overlap increases to almost 100 percent if one of the cores is left spare for the operating system's processes and the process that handle the



(a) 9 MPI processes per node, 1HP for SmartInterrupts



(b) 18 MPI processes per node, 2HP for SmartInterrupts (c) 17 MPI processes per node, 2HP for SmartInterrupts

Fig. 4.7. Communication/Computation Overlap of Point-to-Point Communications

Verbs API calls. This is shown in Figure 4.7(c). Note, that for 17 and 18 processes per node, the increased number of communicating pairs increases the network load as well. This provides enough time to the helper processes, resulting in the improved overlap of 32KB messages.

Scalability

Figure 4.8 shows an analysis of the scalability of MVAPICH-Async and SmartInterrupts. As observed in the previous results, the overlap of MVAPICH is negligible. Hence, it is not represented in this figure. The scalability test is done based on two criteria, message size and the number of processes. The graph reports the overlap for each combination of those criteria. As previously discussed, the overlap of relatively smaller rendezvous messages is expected to be small. So, the message size of 32 KB was chosen to represent them. 256KB messages were chosen to represent larger messages. The figure reports the results with 1, 4, 8, 12 and 16 MPI processes per node. For

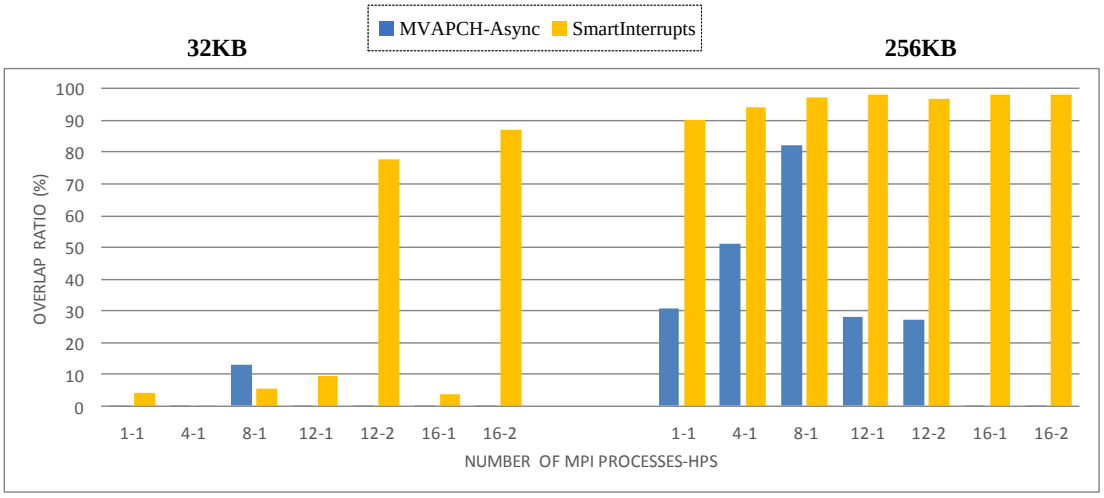


Fig. 4.8. Scalability Analysis of Two-Sided SmartInterrupts

12 and 16 MPI processes, it also shows the results with different number of helper processes. It can be observed that SmartInterrupts scales well with both the parameters of message size and number of processes. Also, in case of 12 and 16 MPI processes, one helper process is enough to achieve close to 100 percent overlap for messages of size of 256KB, but for 32KB messages, a minimum of two helper processes is required for overlap.

Asynchronous Message Progression

With reference to the generalized algorithms in Figure 4.5, the major difference between the overlap and asynchronous message progression micro-benchmarks is that in the former, the timing at the receiver is ended only after completion of the synthetic work. However, in the latter, the timing is stopped as soon as the user data progression is complete, regardless of whether the synthetic work is over or not. For this, the synthetic work is based on the message latency of MVAPICH; that is, its T_{lat}. The synthetic work is varied between 1.5 to 3 times of T_{lat}, with 10% increments. At the end of each iteration, the elapsed time is calculated and the average of the elapsed times across all receiving MPI processes is reported. In the graphs, if the curve of a particular asynchronous progression technique is lower than that of MVAPICH then that is an

indication of asynchronous progress. Also, in the ideal scenario the curve is expected to be linear and have a slope of zero, as that indicates that the progression time remains constant regardless of the synthetic work. Because of the way the asynchronous message progression is interpreted, the synthetic work cannot be less than T_{lat}. Also, synthetic work up to 1.1 times of T_{lat} was already covered with the overlap micro-benchmark. Therefore, for asynchronous message progression, the synthetic work was started at 1.5 times of T_{lat}. The maximum synthetic work is non-critical but should be high enough to illustrate the trends.

Figure 4.9 and Figure 4.10 show the asynchronous progression results of MVAPICH, MVAPICH-Async and SmartInterrupts. MVAPICH exhibits linearly increasing curves for all message sizes as expected. For 9 processes, the results of MVAPICH-Async and Smart Interrupts are identical, except for message sizes of 16KB and 32KB. This is because of continuous polling, which causes the serialization of polling requests at the verbs API handler. For 18 processes, oversubscription continues to be detrimental to the performance of MVAPICH-Async. Whereas, SmartInterrupts' latencies are lower than that of MVAPICH by about a factor of half.

Memory Footprint Analysis

The memory footprint of SmartInterrupts was analyzed by looking at the physical memory usage of the processes. This was done by examining the value of the **VmRSS** field present in the **status** file of each MPI process listed in the proc vitual filesystem. Also, to know how the memory usage scales, it was measured by doing incremental changes to the number of MPI processes per node. The experiments showed the memory usage of SmartInterrupts to be slightly greater than MVAPICH, by a difference of approximately 300KB per process. This difference remained constant regardless of the number of processes per node. Also, from empirical analysis, it was observed that the additional memory requirement scales according to the number of MPI processes

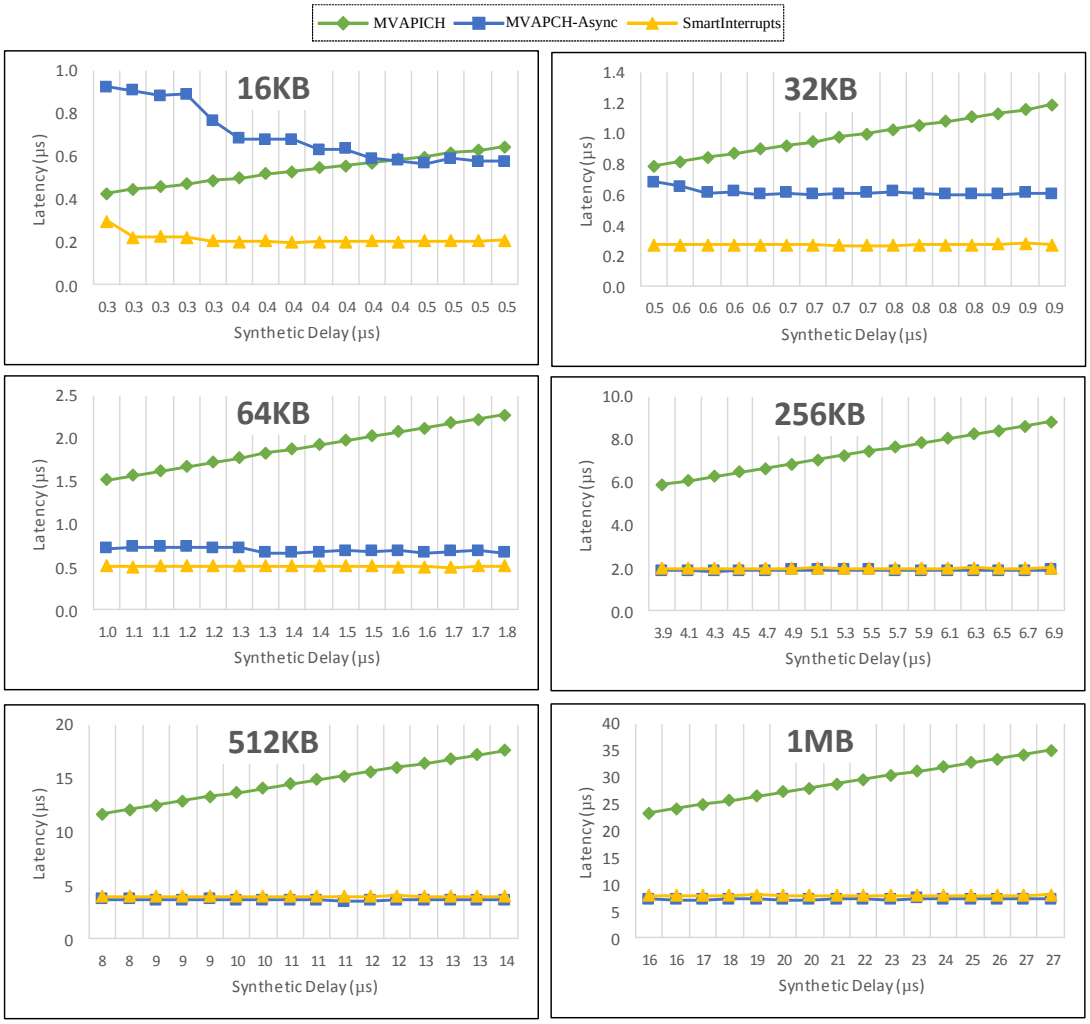


Fig. 4.9. Asynchronous Message Progression for 9 MPI processes per node, 1 Helper Process for SmartInterrupts

per node and not according to the total number of MPI processes. Therefore, even with fat-nodes, the additional memory required would be in the order of a few megabytes, which is quite negligible.

4.5.3 Collective Micro-Benchmarks

In several MPI implementations, collectives are implemented using point-to-point communications. Therefore, overlap and asynchronous message progression micro-benchmarks were designed around non-blocking collectives to see if SmartInterrupts can improve the performance of collective communications. It was observed that SmartInterrupts can indeed

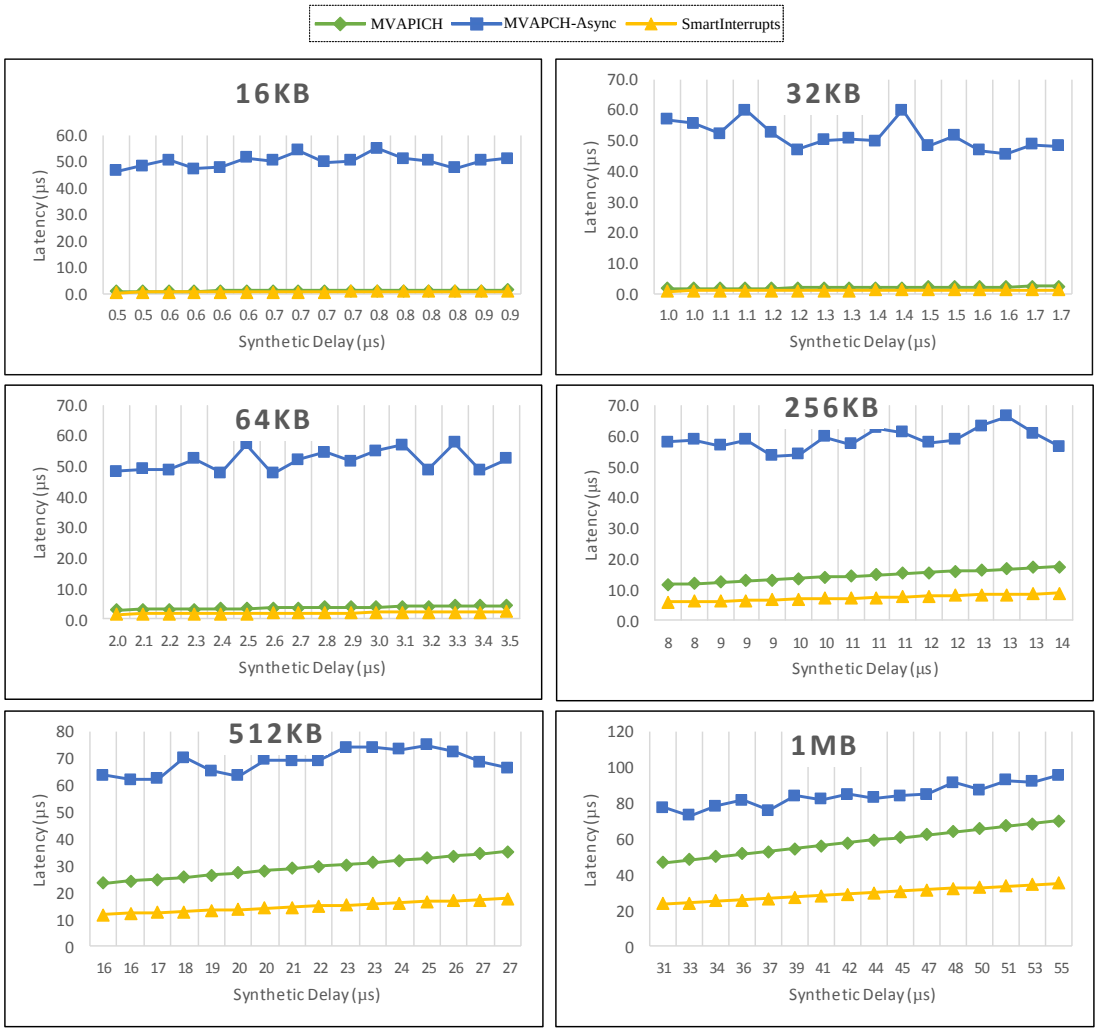


Fig. 4.10. Asynchronous Message Progression for 18 MPI processes per node, 2 Helper Process for SmartInterrupts

improve the performance of collectives, but not all of them. The micro-benchmarks were designed around non-blocking collectives like MPI_Iscatter, MPI_Igather and MPI_Ialltoall. SmartInterrupts performed better for MPI_Ialltoall compared to the other two.

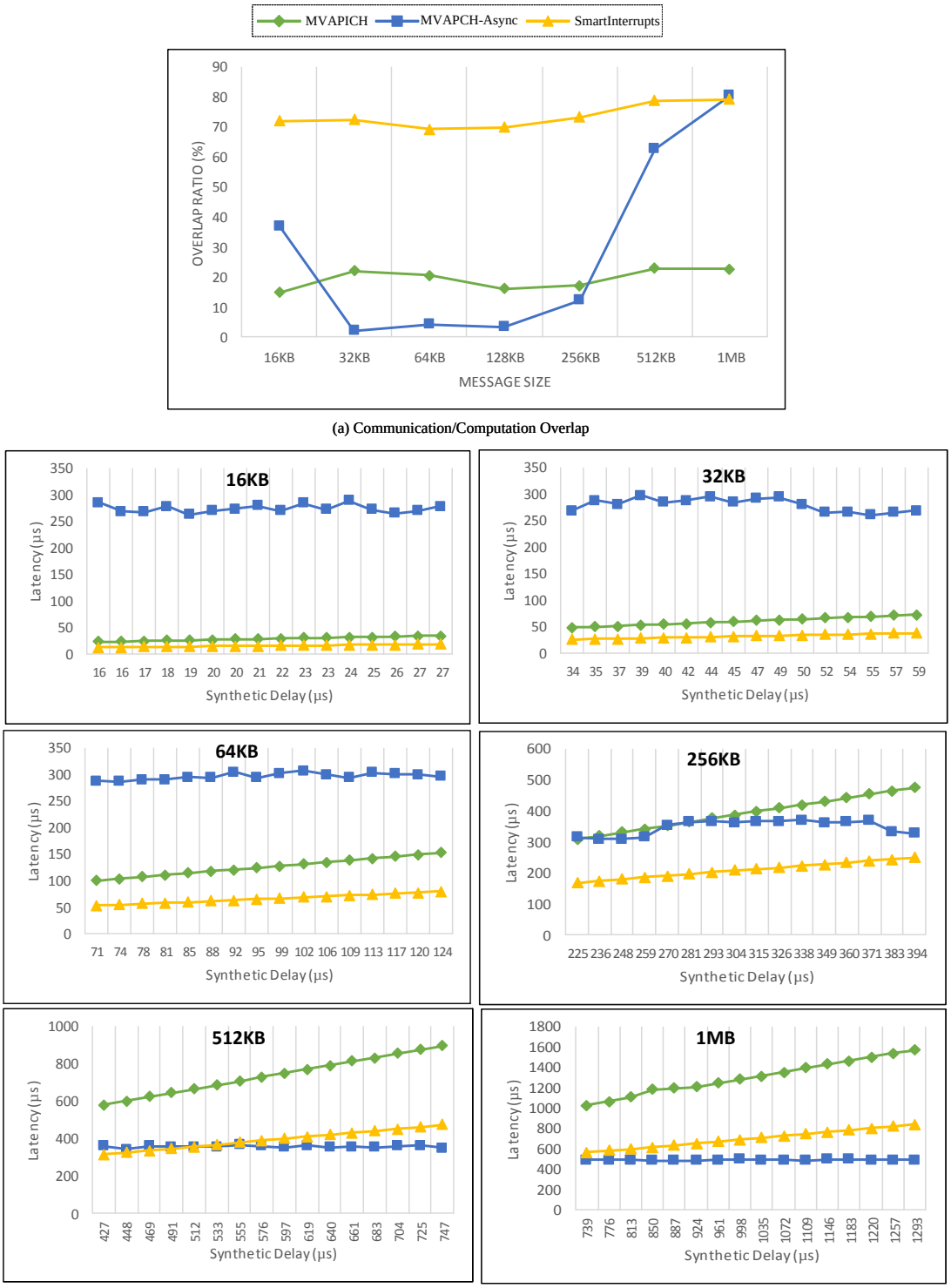
Figure 4.11 shows the algorithm template that was used to design the collective versions of overlap and progression micro-benchmarks. Like their two-sided versions, the major difference between the two micro-benchmarks is the implementation of line (4). In the overlap micro-benchmark for collectives, the synthetic work is varied between 0.1 and 1.1 times of T_{lat} and the

```
1. MPI_Barrier(MPI_COMM_WORLD)
2. Measure Start Time
3. MPI_Ialltoall/MPI_Iscatter/MPI_Igather/MPI_Ibcast
4. Variable synthetic work according to the micro-benchmark
5. MPI_Wait
6. Measure Stop Time
```

Fig. 4.11. Template for Collective Micro-Benchmark Design

latency measurement is ended only after the end of the synthetic work. The formula used for the calculation of overlap is the same as the one used for point-to-point communications, as in Equation (4.2). In the asynchronous progression benchmark, the synthetic work is varied between 1.5 to 3 times of T_{lat} and the latency measurement is ended as soon as the user data from all the peers is available. The rationale behind the amount of synthetic work for both the micro-benchmarks are the same as those stated for their two-sided counterparts. For the overlap micro-benchmark, it is based on empirical measurements. The progression micro-benchmark's synthetic work was chosen with a view to observe trends. For this, the synthetic work cannot be less than T_{lat}. The idea is to have a synthetic work that is slightly greater than T_{lat}, which is then incrementally increased until a trend is observed.

SmartInterrupts' collective communication performance was evaluated on 20 nodes of the cluster mentioned previously. The micro-benchmarks were executed with 18 MPI processes and 2 helper processes on each node. A total of 18 collective operations were performed in parallel and each operation comprised of one process per node. Therefore, each collective operation was performed among 20 MPI processes and 360 of them were used in total. In both the micro-benchmarks, the average latency was measured over 1000 iterations and the results of the first 100 iterations were discarded. Figure 4.12(a) and Figure 4.12(b) respectively show the overlap and progression results for the MPI_Ialltoall collective. Figure 4.12(a) shows that SmartInterrupts consistently improves the overlap of the MPI_Ialltoall collective compared to MVAPICH and MVAPICH-Async. Also, SmartInterrupts shows asynchronous message progression in all cases with significantly better results than MVAPICH-Async up to 256KB messages. For messages of



(a) Communication/Computation Overlap

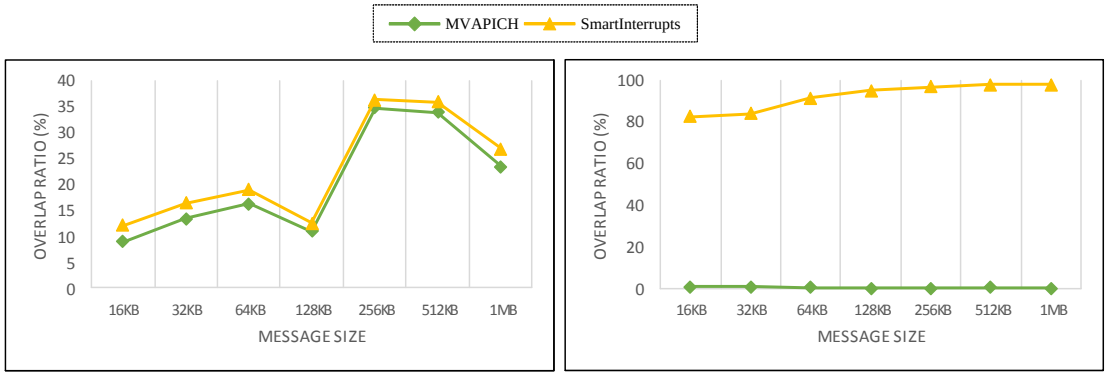
(b) Asynchronous Message Progression

Fig. 4.12. MPI_Ialltoall Overlap and Asynchronous Message Progression Results

size below 512KB, MVAPICH-Async suffers overheads due to oversubscription which affects both the overlap and the asynchronous message progression. However, the considerably larger synthetic delays of 512KB and 1MB messages provide enough time to the polling threads of MVAPICH-Async to asynchronously progress all the messages in time, even with the presence of oversubscription.

The reason why SmartInterrupts is unable to significantly improve the overlap for MPI_Iscatter and MPI_Igather is due to their implementations, which do not provide enough decision-making information to the helper processes. The MPI_Iscatter collective is implemented using a binomial tree algorithm in MPICH and MVAPICH. This scheme may lead to nodes that have both parents and children. These nodes cannot initiate a transfer to their children unless they have received their own message from their parents. Once the rendezvous message transfer from the parent is complete, another call to the progress engine is required to send the messages to the children. SmartInterrupts can successfully assist in asynchronously progressing the parents' messages but currently there is no way of informing the helper processes about the completion of the RDMA Read operation. The same limitation exists with other collectives like MPI_Igather and MPI_Ibcast. In such cases, the performance will be the same or very slightly better than having no progression threads. This can be seen in Figure 4.13(a).

To show that the scheduling algorithm is indeed the reason for the sub-par performance, SmartInterrupts' performance with the default algorithm was compared against its performance with another algorithm that avoids the exchange of aggregate messages. One such algorithm for these collectives is a naïve approach in which MPI_Isend and MPI_Irecv calls are used for each message in the collective. For example, for the MPI_Igather collective with *N* MPI processes, this would mean calling *N-1* MPI_Irecvs at the root process and one MPI_Isend or MPI_Send at the rest of the processes. SmartInterrupts exhibits a good amount of overlap with such an algorithm. The results are illustrated in Figure 4.13(b). Because of its inferior performance due to



(a) Using Binomial Tree Algorithm (b) Using Naïve Algorithm

Fig. 4.13. MPI_Igather Micro-benchmark Results

oversubscription and because the point of this experiment was to find if SmartInterrupts can improve the overlap over the default, no asynchronous message progression case, these figures do not show the results of MVAPICH-Async.

4.5.4 Application Results

To evaluate SmartInterrupts in a practical scenario, it was tested by using the **Scalar Penta-diagonal solver (SP)**, which is one of the applications in the **NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB)** [57]. NPB is a set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. NAS-SP is one of the pseudo-applications. It performs a synthetic CFD problem by solving multiple, independent systems of non-diagonally dominant, scalar, penta-diagonal equations. The NAS-SP application is available with 5 classes of problems, S, A, B, C, D and E. E being the largest problem class, requiring the maximum amount of computation resources. Also, this application requires the number of MPI processes to be a perfect square. This application uses MPI_Isend and MPI_Irecv two-sided calls, and MPI_Bcast and MPI_Reduce collectives. The buffer size specified in the collectives are small and they do not use rendezvous messages. On the other hand, the buffer size

specified in the two-sided calls is dependent on the number of MPI processes. Hence, the possibility of using rendezvous messages.

SmartInterrupts was experimented with NAS-SP using different problem classes and different number of MPI processes, and the results were compared to those of MVAPICH. In all the executions, a very small number of CPU cores (not more than 4) were left spare. The reason behind this was to simulate a real world scenario where the nodes are packed densely with MPI processes, with very few spare cores available. Coincidentally, this also puts maximum stress on the helper process and helps to extensively evaluate the performance of SmartInterrupts. Therefore, running NAS-SP using MVAPICH-Async was pointless as it was guaranteed to perform worse than the other test subjects due to oversubscription. Figure 4.14 compares the application execution results

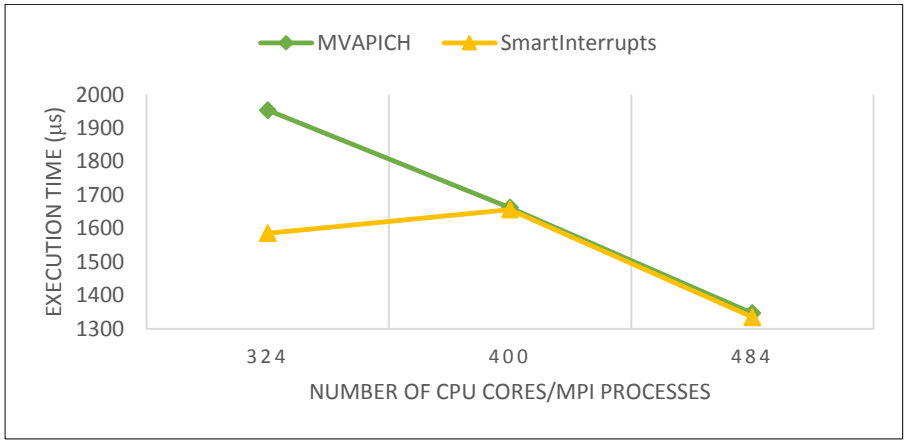


Fig. 4.14. NAS-SP Results (Class E)

of SmartInterrupts and MVAPICH with strong scaling. In strong scaling, the problem size remains fixed and the number of processing elements are varied. This figure shows the results for the **problem class E** with 324(18^2), 400(20^2) and 484(22^2) MPI processes, distributed evenly on 21, 25 and 31 compute nodes respectively. Throughout the experiments with SmartInterrupts, two helper processes were used per node. The results show that the proposed approach can indeed make a difference in a practical scenario. The most interesting result here is the one with 324 MPI

processes, in which it shows an improvement of about 19 percent. In fact, SmartInterrupts' execution time in this case is lower than MVAPICH's and its own execution time with 400 MPI processes. However, SmartInterrupts shows very little improvement with 400 and 484 MPI processes. This is because the application scales strongly and the size of the rendezvous messages is inversely proportional to the number of MPI processes. With 400 and 484 MPI processes, the size of the messages drop below the eager threshold, so they end up being transferred eagerly. Eager messages get overlapped naturally, therefore, SmartInterrupts' involvement becomes negligible. Hence, the relatively small performance improvement compared to 324 MPI processes. The same behavior was observed with Class D as well.

4.6 Summary

This chapter presented the design, implementation and performance evaluation of a node-wide asynchronous message progression technique called SmartInterrupts. Unlike protocol improvement methods, SmartInterrupts behavior is completely deterministic, meaning that there is no randomness involved. The action will always be preceded by a predetermined set of events. Unlike, hardware-based approaches, it does not require specialized hardware and only requires a few spare CPU cores. Among host-based approaches, the most common is asynchronous message progression, which may be based on polling or interrupts. Their advantages and disadvantages are mentioned in this chapter and discussed in depth in Section 3.1.2. SmartInterrupts is a hybrid approach that harnesses the strengths of both polling and interrupt based approaches while completely avoiding their disadvantages.

The proposed approach uses the sender initiated RDMA Read based rendezvous protocol to leverage its natural overlap potential when the sender arrives first. The progression is performed by interrupt threads which are triggered into action by the helper processes when the message matching parameters are met. Each MPI process has its own interrupt thread and several of these

interrupt threads may be associated with a single helper process. This design was implemented on MVAPICH and its performance was evaluated using two-sided and collective micro-benchmarks, and the NAS-SP application. The results were compared with those of the default MVAPICH with no asynchronous progression and with its polling based asynchronous progression.

The two-sided micro-benchmarks were designed to test the latency overhead, communication/ computation overlap, asynchronous progression, scalability and memory footprint. It was observed that the overhead incurred by SmartInterrupts is negligible and that close to 100 percent overlap can be achieved for most message sizes that involve a rendezvous protocol. The progression results showed significantly lower message latencies compared to synchronous progression and in most cases, the latencies remained nearly constant for a particular message size. This approach was found to be scalable and cast an insignificant amount of memory footprint. The collective micro-benchmark was designed to assess the improvement in overlap that can be obtained using SmartInterrupts. It was observed that the current implementations of collectives do not expose enough decision-making information for SmartInterrupts to act upon. However, it was found to be effective for MPI_Ialltoall, and MPI_Igather with a modified algorithm. Finally, it was tested with the NAS-SP application to evaluate its performance in a practical scenario, and the results were positive. **These results emphatically answer the second research question posed in Section 1.2.**

This chapter presented a novel approach called SmartInterrupts for point-to-point communications. A similar approach can be used to improve the performance of RMA communications as well. The next chapter discusses the design and implementation of the extension of SmartInterrupts to one-sided communications.

Chapter 5

Node-Wide Asynchronous Message Progression Technique for One-Sided Communication

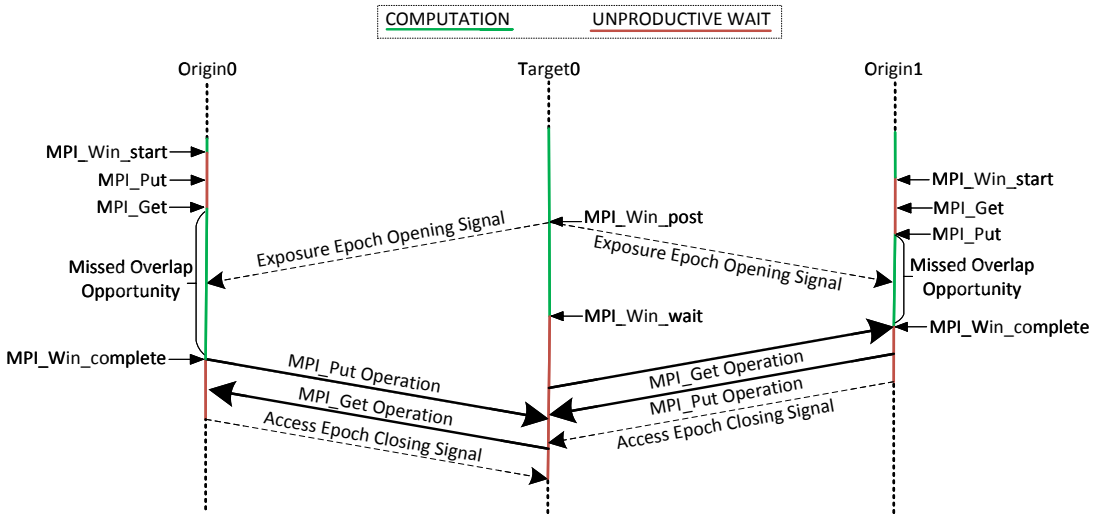
Chapter 4 discusses the design, implementation and performance evaluation of SmartInterrupts for point-to-point communications. It is an asynchronous message progression technique which utilizes the advantages of polling and interrupt based approaches. This chapter describes such an asynchronous progression approach but for one-sided communication[1]. As a reminder, in one-sided communication, the peer that issues the RMA operations such as MPI_Put and MPI_Get is called the origin and the peer on which these operations are performed is called the target. Unlike two-sided communication, the successful transfer of an RMA message does not require a matching call from both the peers. In fact, the target is not required to issue any calls for the RMA operations. However, an RMA operation cannot be initiated by an origin until the target has opened its exposure epoch. Therefore, synchronizations are performed among the peers so that they can become aware of the opening and closing of each other's epochs. The MPI standard [54]

[1] The code to add the shared buffers such as ICB and IRB, and to expose interrupt information through them was contributed by Dr. Judicael Zounmevo. Development of the interrupt thread, the helper process and the interrupt handler kernel module was done by the author.

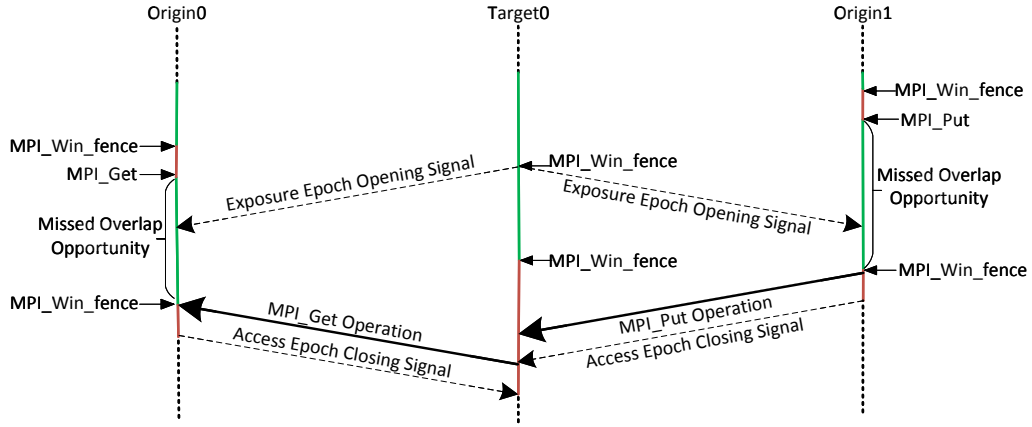
specifies two types of synchronizations, namely, active target and passive target. Active target synchronizations can be further classified into Fence and GATS. Similarly, passive target synchronizations can be classified into shared lock epochs and exclusive lock epochs. The details of one-sided communication can be found in Section 2.3.3. Chapter 3 discusses the inefficiencies associated with one-sided communications and presents a survey of the literature that attempt to address these inefficiencies. Specifically, these inefficiencies inhibit the natural communication/ computation overlap that is expected from one-sided communication. This chapter investigates the third research question posed in Section 1.2 by describing an approach that improves the overlap of one-sided communications by using a similar technique as the one discussed in chapter 4. Due to the similarity in design and principle, the work proposed in this chapter is also referred to as SmartInterrupts. The following sections present the motivation behind the proposed technique, followed by its design and implementation, and finally its performance evaluation by means of micro-benchmarks and applications.

5.1 Motivation

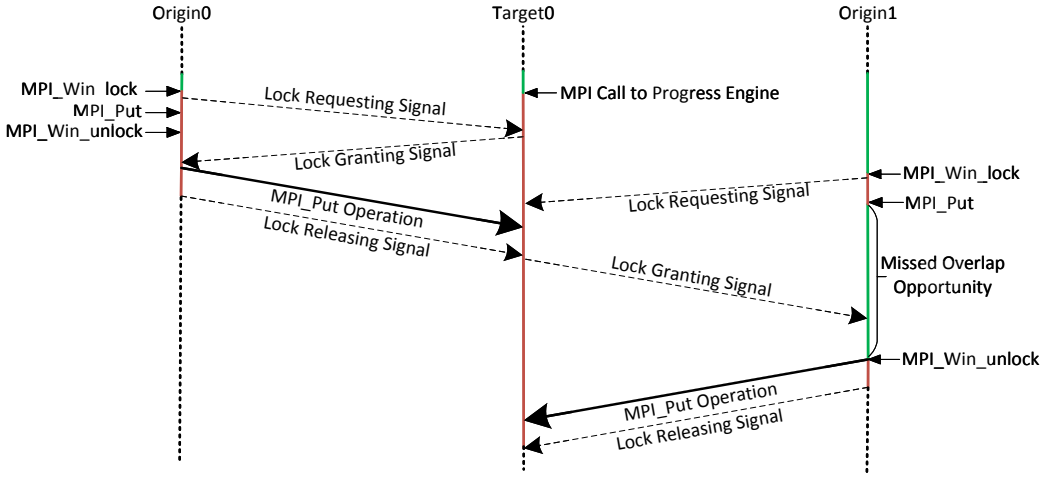
As opposed to two-sided communication, the MPI standard does not define the blocking or non-blocking behaviour of RMA synchronizations. Therefore, the MPI implementations are free to decide the synchronization behaviours as they see fit. Popular MPI implementations like MPICH [55], MVAPICH [56] and OpenMPI [60] provide non-blocking epoch opening calls and blocking epoch closing calls. Blocking synchronizations lead to the inefficiency patterns discussed in [33, 45, 88] and in Section 3.2.1. Non-blocking synchronization calls are imperative to facilitate any kind of overlap and several papers propose exactly that [8, 78, 88]. However, overlap can still be hampered even with the use of non-blocking RMA synchronizations. For instance, consider Figure 5.1(a) which shows an example of a GATS epoch. This example uses non-blocking epoch opening,



(a) GATS Epoch



(b) Fence Epoch



(c) Exclusive Lock Epoch

Fig. 5.1. Inefficiencies with Non-blocking RMA Synchronizations

so the origins open their access epochs, issue the RMA operations and proceed to their computations. Although the RMA operation calls have been made, the message transfer cannot initiate until the target opens its epoch. But when the exposure epoch opens, the origins are busy in their computations so the message transfers cannot happen until the next call to the progress engine. Since there is no call to the progress engine after the arrival of the exposure epoch opening signals and before the MPI_Win_complete calls, this ultimately leads the entire communication to be deferred to the MPI_Win_complete calls of the origins, causing unproductive waits at all the peers. A similar inefficiency is possible in a fence epoch as well and it is illustrated in Figure 5.1(b) In this case, all the communications are deferred to the epoch closing MPI_Win_Fence calls, leading to no communication/computation overlap at any of the peers.

Passive target synchronizations have similar inefficiencies as well. In passive target synchronizations, the target is not required to open its epoch. Once the window objects are exchanged among the associated targets and origins, the origins are free to perform operations on the remote memory of the targets. As mentioned in Section 2.3.3, the passive target synchronization entails a lock/unlock mechanism, where a lock can be either acquired exclusively or be shared. The target does not issue any calls but acts as the intermediary between the origins and grants them the lock to perform RMA operations. It does this implicitly at the middleware when the progress engine is active. Figure 5.1(c) illustrates a scenario in an exclusive lock epoch where Origin0 calls the locking function ahead of Origin1. The target keeps the progress engine active by calling a blocking MPI communication function such as MPI_Wait or MPI_Barrier. Because of its early arrival, Origin0 is granted the lock first. So it performs its operation and releases the lock. Origin1 on the other hand, is involved in a long computation, so it cannot immediately initiate its RMA operation even though the target has already granted it the lock. This not only causes an unproductive wait at Origin1 but may also lead to the propagation of this wait to other origins that are waiting for the lock. In case of a shared lock epoch, any origin is free to perform operations after calling the

MPI_Win_lock function, hence it is unlikely to have communication/computation serializations in this case. This is demonstrated through an example in Figure 5.2, where even though Origin1 issues the MPI_Win_lock call first and does not call MPI_Win_unlock until much later, Origin0 can still issue RMA operations and progress them immediately. Therefore, in shared lock epochs, some degree of overlap can generally be expected if an RMA operation call is followed by a computation.

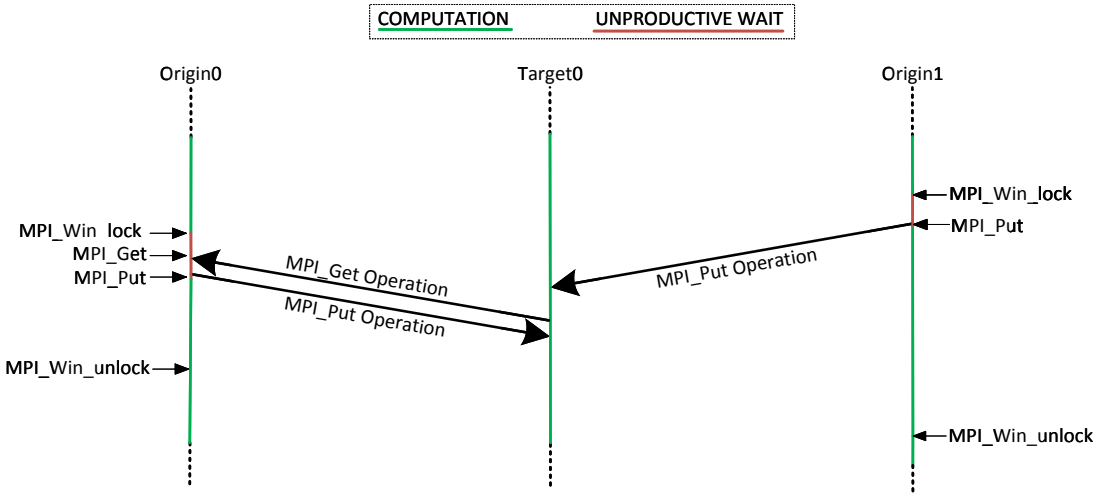


Fig. 5.2. Illustration of Communication/Computaiton Overlap with Shared Lock Epochs

Casper [73] has the ability to deal with the inefficiencies described above, however, there are certain drawbacks associated with it. First, in Casper, the ghost processes perform the RMA operations on behalf of the MPI processes. This mandates the creation of a very large shared buffer between the MPI processes and their ghost process because the application's entire RMA data needs to be shared. Wasting a lot of CPU cores for the ghost processes is not practical so there may be instances where the ghost process is mapped to one NUMA node and its MPI processes are mapped to another. This may cause a massive amount of data to be transferred through the inter-socket interconnect that connects the CPUs, adding an extra layer of latency above the message latency of inter-node communications. SmartInterrupts also uses a shared memory between the MPI processes and its helper process but the amount of data shared is in the order of bytes, as opposed to entire RMA messages in Casper's case, where the message size is dictated by the

application. Another problem with Casper is that it forces the MPI processes to rely on their ghost processes, that is, even if an MPI process is free to progress its own RMA communications, it cannot do so because all its synchronizations and RMA operations are automatically redirected to its ghost process. This is inefficient because it causes a wastage of CPU cycles at the MPI process and redirections are obviously slower than having the MPI process call its progress engine itself. In SmartInterrupts, the progression thread is not activated if a call to the progress engine is active at its MPI process.

Currently, there is no practical overlap technique that works well for both two-sided and one-sided communications. Polling based asynchronous message progression [34] is a solution but it suffers from the issues of inefficient resource utilization and oversubscription (Section 3.1.2). SmartInterrupts' effectiveness with two-sided communication has already been discussed in Chapter 4. This chapter leverages the principle behind two-sided SmartInterrupts to improve the overlap of one-sided communication. At this moment, there are two separate implementations for one-sided and two-sided communications. However, it is possible to unify the two designs.

5.2 Design of SmartInterrupts for One-Sided Communications

Although the semantics of two-sided and one-sided communications are different, the issues that inhibit communication/computation overlap in them are similar in principle. The first such issue is the significant time difference between the calling of matching MPI functions. In the rendezvous protocols of two-sided communication, this translates to a significant delay in calling the matching send or receive function by the sender or receiver respectively. In one-sided communication, this is equivalent to a non-timely issuance of a synchronization call. For two-sided sided communication, some approaches have been discussed in Chapter 2 to address this issue but such approaches are not scalable. In general, this is an application specific issue and apart from using non-blocking calls, not much can be done to mitigate the inefficiencies caused by them.

The second issue is the delayed acknowledgement of a control signal that fulfills the progression criteria of pending messages at the middleware. This delay is caused when the control signal arrives while the MPI process is busy in a computation. For instance, in a sender-initiated RDMA Read based rendezvous protocol, if the receiver arrives early then the message should ideally be progressed immediately when the RTS arrives at the receiver. However, if the RTS arrives while the receiver is busy in a computation, then the message progression is delayed until the next call to the progress engine. Thus, the opportunity to overlap the communication with the computation is lost. As discussed in the previous section, the same issue can happen in one-sided communication when the epoch opening signal in Fence or GATS, or the lock granting signal in an exclusive lock epoch arrives when the origin is busy. Asynchronous message progression approaches work well for this issue because even though the application threads of an MPI process might be busy, the progression thread remains vigilant to the incoming control messages and immediately acknowledges them when they arrive.

SmartInterrupts is also an asynchronous message progression technique, whose two-sided design and implementation is discussed in Chapter 4. Because of the similarity in the problem statement and the solution, it was hypothesized that the design proposed in Chapter 4 could be extended to one-sided communication as well. Therefore, to address the overlap inhibiting scenarios discussed in the previous section, the design objectives followed for the development of one-sided SmartInterrupts are the same as those of two-sided SmartInterrupts. Similar to Chapter 4, the following design objectives are considered for one-sided communication:

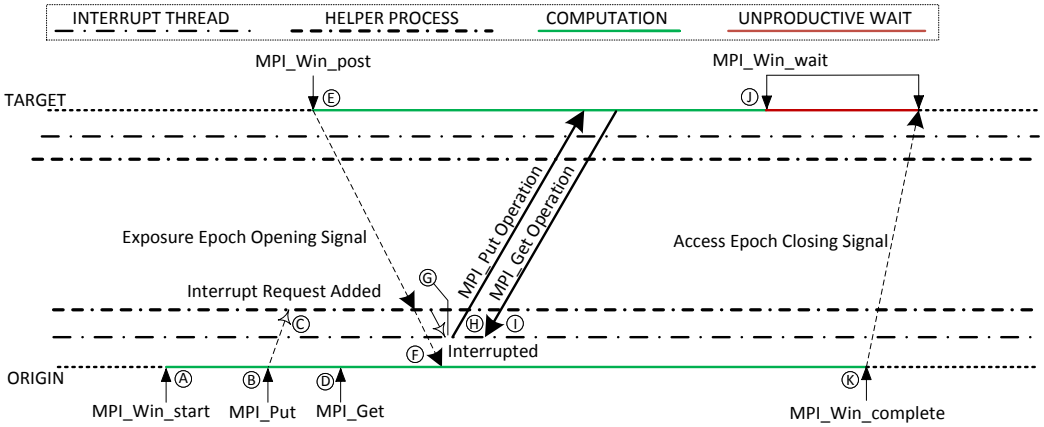
1) Hardware interrupts are expensive. Therefore, a mechanism is required to minimize or eliminate hard-interrupts if possible.
2) The resource efficiency of interrupt based asynchronous message progression is desired in the design, but a mechanism is required to selectively trigger interrupts due to the overheads associated with futile software interrupts.

3) Polling based asynchronous message progression is more responsive to incoming messages; however, it is impractical to waste one CPU core per MPI process for polling threads. Therefore, a mechanism is required to:
    a) Associate multiple MPI processes with a single polling thread.
    b) Share information about the incoming control signals of multiple MPI processes with the associated polling thread. This information is limited to the address space of individual MPI processes. However, the polling thread would likely have its own address space or share the address space with one other MPI process.

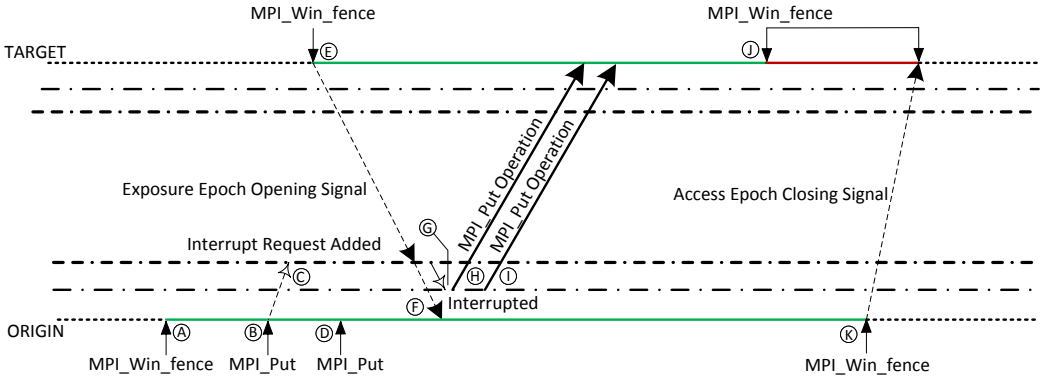
The result of these objectives is a design that is very similar in principle to two-sided SmartInterrupts. Like the two-sided design, asynchronous message progression in this design is performed by an interrupt thread and the interrupts to this thread is generated by a polling process known as the helper process. Each of these helper processes may be associated with several MPI processes and each MPI process is augmented with its own interrupt thread. The following subsections describe the working mechanism of one-sided SmartInterrupts and its core components.

5.2.1 Asynchronous Message Progression Mechanism

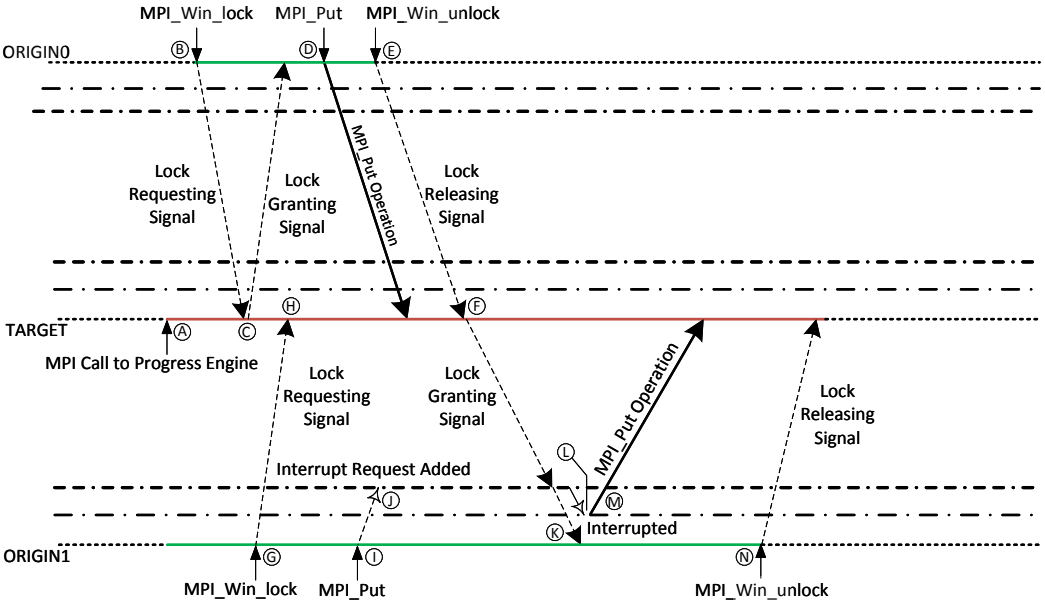
Figure 5.3(a) illustrates the data-movement and signaling with SmartInterrupts in a GATS epoch. This figure shows SmartInterrupts' solution for the issue discussed in the motivation section, where the RMA communications end up being deferred due to a busy origin. This approach avoids hardware interrupts and relies entirely on software interrupts, therefore it addresses Design Objective 1. To address Design Objectives 2 and 3(b), two shared buffers are created between each MPI process and its helper process. One of these buffers are manipulated locally by the origin to enable and disable the interrupt mechanism, and the other holds the control signal information that can be manipulated remotely by the target . These buffers are called Interrupt Request Buffer (IRB)



(a) GATS Epoch



(b) Fence Epoch



(c) Exclusive Lock Epoch

Fig. 5.3. Data Movement and Signals in One-Sided SmartInterrupts

and Interrupt Control Buffer (ICB), respectively. When the origin issues its RMA operation at (B) and realizes that the target has not opened the exposure epoch yet, then it queues the RMA communication for later and writes to the IRB at (C) to request help from its helper process. This essentially enables the interrupt mechanism by informing the helper process that an interrupt should be triggered to the asynchronous thread when the exposure epoch opening signal arrives.

When the target calls the MPI_Win_post function at (E), it informs the origin about the opening of the exposure epoch and writes to the ICB at (F). Without an asynchronous mechanism, this opening of the exposure epoch would not be acknowledged by the origin until the end of the computation. In SmartInterrupts, however, the helper process continuously polls on the data in the ICB. When the helper process detects the exposure epoch opening signal, it checks if the origin has requested for interrupts. When the two conditions are satisfied, the helper process triggers an interrupt at (G) to progress the pending MPI_Put and MPI_Get RMA operations. Figure 5.2(b) illustrates a similar solution for the fence epoch.

Figure 5.3(c) illustrates the behavior of SmartInterrupts in a typical exclusive lock epoch scenario. As previously stated, in an exclusive lock epoch, the target is not required to issue RMA specific MPI calls but still has to participate implicitly to distribute the locks. Origin1 is busy when the lock granting signal arrives at (K). Therefore, in the absence of asynchronous message progression support, the MPI_Put operation issued at (I) would not be progressed until the end of the computation at (N). However, in SmartInterrupts, the helper process remains informed of the arrival of such signals; therefore, it triggers an interrupt at (L) to the origin's progression thread since a request for interrupts was made earlier at (J). This interrupt causes the timely progression of the MPI_Put operation at (M) instead of (N).

5.2.2 Core Components

The similarity in the designs of two-sided and one-sided SmartInterrupts leads to the similarity of components as well. The core components of this design consist of the progression threads, the Interrupt Handler kernel module, the helper processes and two shared buffers (ICB and IRB) between each MPI process and its helper process. Each MPI process consists of one application thread and one asynchronous progression thread which is oversubscribed with the main application thread. In this case, oversubscription incurs negligible performance penalty because the progression thread is based on interrupts, which does not poll periodically.

The Design Objective 1 is achieved by the Interrupt Handler kernel module which enables the complete avoidance of hardware interrupts and provides the means to support SmartInterrupts' interrupt mechanism. It creates a virtual file in the proc filesystem and defines the behavior of the read() and write() system calls associated with that file. The interrupt threads call this read() function to go to sleep and the helper processes call the write() function to awaken the sleeping progression threads. It is important to note that the helper processes do not deal with the NIC directly and do not issue network API calls. This is the job of the application and progression threads.

Each MPI application can be launched with a user-defined number of helper processes. This number can be set as an environment variable before executing the application. However, at least one per node is required to expect progression from the interrupt thread. Also, in most cases, one helper process should be enough to handle asynchronous progression for the entire node. For optimum performance, each helper process needs to be dedicated to a CPU core because of the existence of a busy loop. Details of certain components can be found in Section 4.2.1.

5.3 Implementation of One-Sided SmartInterrupts

NewRMA was implemented on MVAPICH2 and it incorporates the designs of [86] and [88]. [88] proposes non-blocking epochs to avoid any waiting during epoch opening or closing, whereas, [86] proposes a scheduling scheme to improve the overlap of intra-node RMA communications by exploiting the residual overlap potential of inter-node communications. Since these approaches have already been documented to perform better than the existing implementations, SmartInterrupts was implemented on top of NewRMA.

5.3.1 Shared Buffers

Before calling any MPI function, each process of the MPI application must call **MPI_Init** to initialize the MPI execution environment. In NewRMA, the peers signal the opening and closing of epochs by performing an RDMA Write to a remote buffer at the peer. During MPI_Init, NewRMA exchanges the addresses of the buffer on which the control signals will be written. For SmartInterrupts, NewRMA was modified to exchange the address of another 1-byte buffer. This buffer is the ICB and it is shared with its helper process. When a target exposes its epoch or wants to grant the lock to an origin, it manipulates the ICB by performing an RDMA Write to it. The ICB helps to achieve the Design Objective 3(b) by enabling the helper process to know about the incoming control signals. The data contained in the ICB is referred to as Interrupt Control Data (ICD). The default value of each ICD is zero and the target sets it to a non-zero value to indicate the arrival of a control signal.

Another shared buffer, called Interrupt Request Buffer (IRB) was created to send requests to the helper processes. This buffer addresses the Design Objective 2 and provides a mechanism to selectively generate interrupts. The IRB is an array of 8-byte elements called Interrupt Request Data (IRD), where each IRD represents the interrupt request status of its MPI process. A non-zero

IRD indicates that its origin expects interrupts from its helper process. During an RMA operation call, if the origin finds the exposure epoch of its target to be closed or if the target has not granted the lock to the origin, then it increments the ICD by one. In case of fence and GATS, subsequent RMA operations to the same exposure epoch has no effect on the IRD, however, if some other exposure epoch is found to be closed, then the IRD is again incremented. Similarly, in case of exclusive lock epochs, no changes are made to the IRD if the lock of a particular target is already acquired. The reason behind this is that once the origin has acknowledged the opening of the exposure epoch or the granting of the lock, any subsequent RMA operation will not require SmartInterrupts' support and can be progressed immediately. Note, that each MPI process has its own ICD and IRD.

5.3.2 Helper Process and Interrupt Thread

Each helper process consists of a loop which iterates until none of its MPI processes is active anymore. Inside this loop, the ICD and IRD of each associated MPI process is matched to generate interrupts. A non-zero ICD and IRD for a particular MPI process means that it has requested for interrupts and its control signal has arrived. Therefore, an interrupt should be generated if both ICD and IRD are non-zero. However, before the interrupt is generated, another important check needs to be made. If the progress engine is already active in the MPI process, then it would most likely see the opened exposure epoch and progress the pending RMA communications. Triggering an interrupt in this case would serve no useful purpose and add unnecessary overheads. Therefore, there is another variable that is shared between each MPI process and its helper process. This variable contains information about the progress engine semaphore. The interrupt thread is awakened if a match is found (non-zero ICD and IRD) and the progress engine lock is available. Once the interrupt is triggered, the ICD is reset to make room for a new control signal. This is because, once the origin is aware of the opened exposure epoch or granted lock, subsequent RMA

operations to the same epoch will always succeed. On the other hand, the IRD is decremented by the number of exposure epochs that were found to be open and the number of locks that were granted in the last interrupt.

With this approach, there is one scenario where an interrupt may be wrongly triggered. This happens when an MPI process is the origin for multiple targets. Particularly, when the ICD is manipulated by a target which is not the same as the one being expected by the origin. In this situation, the interrupt would be triggered but the call to the progress engine would be short and the interrupt thread would immediately go back to sleep again. This scenario required a careful consideration of the trade-offs. The alternative was to create individual ICDs for all the other MPI processes and share the remote addresses and keys with each other. This approach would have improved the accuracy of interrupt generation but substantially increased the memory usage and the initial inter-process communication.

The amount of shared information in the one-sided implementation of SmartInterrupts is much smaller than its two-sided implementation. This is because the size of each ICD in the two-sided implementation is 64 bits; whereas, in the one-sided implementation, it is just one bit. Also, MPI applications generally tend to use many more point-to-point MPI calls than RMA epochs, so the inter-process communication between the MPI process and its helper process is expected to be much lower. Because of these reasons, the data locality and the relative mapping of the MPI and helper processes is not as important. However, this implementation uses the same mapping scheme as the two-sided implementation. The details can be found in Section 4.3.

5.4 Performance Evaluation and Analysis

This section presents the performance evaluation of the one-sided implementation of SmartInterrupts. The evaluation was performed by using the implementation as a middleware to

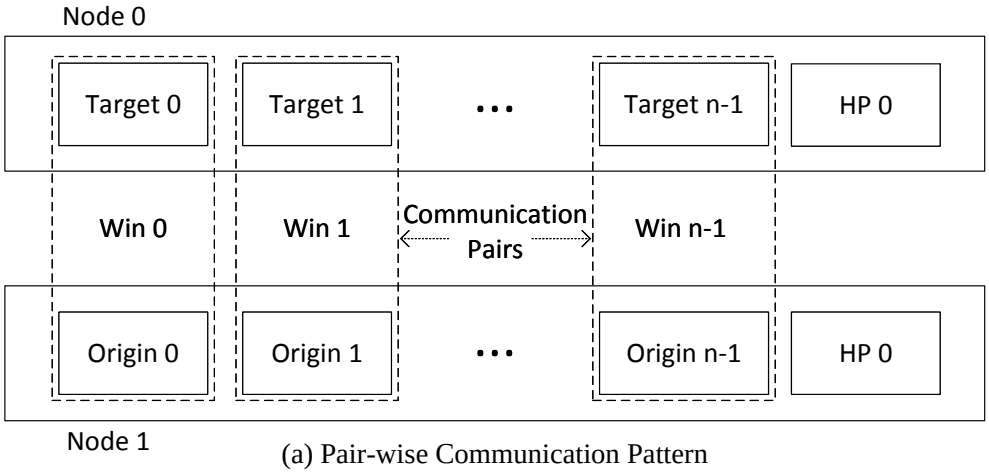
test several RMA micro-benchmarks and one scientific application. These micro-benchmarks and application were then executed on other solutions to assess the effectiveness of the proposed design.

5.4.1 Fence and GATS Micro-Benchmarks

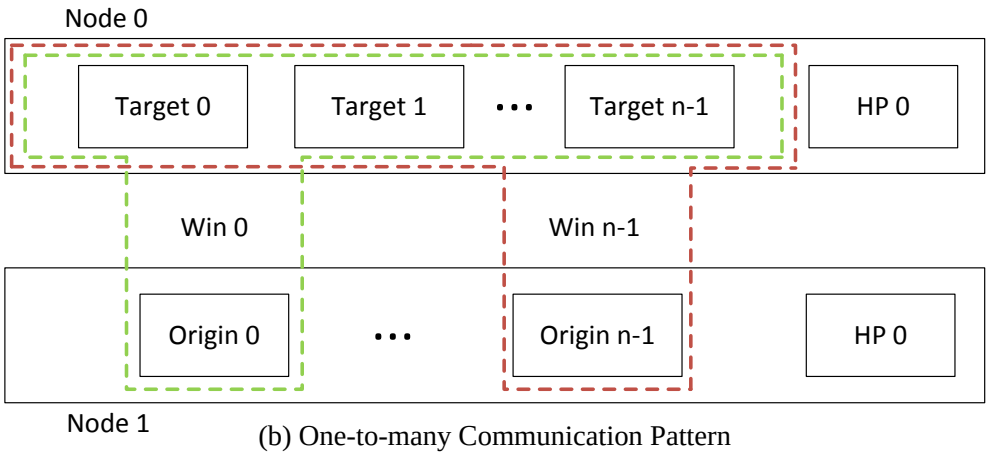
Experimental Setup

The experimental setup for the one-sided micro-benchmarks consists of two nodes of the same cluster that was used for the experiments in Chapter 4. The details of its hardware and software can be found in Section 4.5.1. SmartInterrupts is designed for inter-node communications, therefore, a minimum of two nodes have to be involved. Also, increasing the number of origins per node increases the stress on the helper processes. So, to put maximum stress on the helper processes, all the origins were confined to a single node. The position of the targets is not critical. Hence, they were spawned on the other node. The evaluation was performed on the criteria of latency overhead, communication/computation overlap and memory footprint.

For all micro-benchmarks, two types of communication schemes were tested, pair-wise communication and one-to-many communication. In both the communication schemes, all the MPI processes of one node were designated as the origins and the MPI processes of the other node were designated as the targets. The pair-wise communication scheme is illustrated in Figure 5.4(a). In this scheme, each RMA window object is associated with exactly one origin on one node and exactly one target on the other node, and that origin communicates exclusively with its target. In the one-to-many scheme, each window object is associated with one origin on one node and all the targets on the other node. In other words, each origin performs RMA operations on all the targets that are associated with its window. So, all the origins communicate with all the targets but no two origins share the same window. This scheme is illustrated in Figure 5.4(b).



(a) Pair-wise Communication Pattern



(b) One-to-many Communication Pattern

Fig. 5.4. Communication Patterns Used in One-Sided Micro-Benchmarks

In one-sided communication, it is possible to have as little as one target and one origin associated with a window, and it is also possible to have several targets and origins associated with a single window that are communicating at the same time. This creates a variability on the behavior of SmartInterrupts and the network load. In the pair-wise communication scheme, only a single interrupt is triggered per access epoch, whereas, the one-to-many scheme will require several interrupts per access epoch. Also, the network load with the one-to-many scheme will be much higher than the pair-wise scheme. Therefore, the two communication schemes along with the

different numbers of origins and targets help in the better analysis of SmartInterrupts' performance and overheads.

To study the effectiveness of the proposed approach, the latency overhead and overlap micro-benchmarks were designed with both Fence and GATS epochs. Also, both the epochs were tested with two different RMA operations, MPI_Put and MPI_Get. Before the start of each epoch, the origins and targets were synchronized globally using MPI_Barrier. This is required to ensure that all the communication calls are issued together. Not doing so can cause an irregular load on the network, leading to unpredictable and therefore, unreliable results. The experiments showed similar results for both the targets and the origins. However, this thesis presents the results for the targets only. This is because the target cannot close its epoch until all its origins have closed their own epochs, causing it to suffer the same unproductive waits as its origins.

As mentioned earlier, SmartInterrupts' design was implemented on top of NewRMA [86, 88], which has been shown to perform better than the existing MPI implementations like MVAPICH. Therefore, the results of NewRMA was used as the baseline to evaluate SmartInterrupts. SmartInterrupts' asynchronous progression performance was tested against polling based asynchronous progression, which is supported by NewRMA. For convenience, they are referred to in this document as NewRMA, NewRMA-Async and SmartInterrupts. Casper is available to be used as an extension over existing MPI implementations, however, it could not be experimentally compared with SmartInterrupts because it was found to be incompatible with NewRMA.

Latency Overhead

Figure 5.5 shows the algorithms used at the origins and the targets to measure the latency overhead. In pair-wise communication, the loop at the origin, between lines 4-6 iterates only once because the origin only communicates with one other target. For this and all the following micro-benchmarks, the algorithms were executed 1000 times. Also, the micro-benchmarks used messages

ORIGIN:	TARGET:
1. MPI_Barrier(MPI_COMM_WORLD) 2. Measure Start Time 3. MPI_Win_fence/MPI_Win_start 4. Foreach(target in Targets){ 5. MPI_Put/MPI_Get 6. } 7. MPI_Win_fence/MPI_Win_complete 8. Measure Stop Time	1. MPI_Barrier(MPI_COMM_WORLD) 2. Measure Start Time 3. MPI_Win_fence/MPI_Win_post 4. MPI_Win_fence/MPI_Win_wait 5. Measure Stop Time

Fig. 5.5. Template for One-Sided Latency Overhead Micro-Benchmark

in the range of 16KB to 1MB. The results of the first 100 iterations were discarded to account for cache warm-up and the average duration of the next 900 epochs was recorded. The average was first recorded locally at each origin and target, followed by a calculation of the global averages for all the targets and origins. The values reported here are derived from the global averages.

Since both the asynchronous approaches, SmartInterrupts and NewRMA-Async, work on top of NewRMA, the average epoch durations of NewRMA was chosen as the baseline to measure the latency overheads ($T_{overhead}$). The latency overhead is the difference between the epoch durations of the asynchronous message progression approaches and NewRMA. Figure 5.6 shows the latency overheads of SmartInterrupts and NewRMA-Async for different combinations of MPI processes, RMA synchronizations and communication schemes. SmartInterrupts shows a negligible amount of overhead in all the results. However, the same cannot be said for NewRMA-Async's performance. With 8 MPI processes per node, NewRMA-Async suffers no oversubscription so its overhead becomes comparable to SmartInterrupts for larger messages. For smaller messages, its overhead is high due to the contention for the progress engine lock between the polling thread and the application thread. In Figure 5.6(c), its overhead for 512KB and 1MB messages leans slightly to the negative side due the very small and comparable execution times of non-blocking calls and the message latency of control signals. It is possible that the control signal may not arrive at the origin before the first iteration of line 5. If it does not, then in NewRMA, the progression gets delayed to the next call to the progress engine. However, with polling based asynchronous

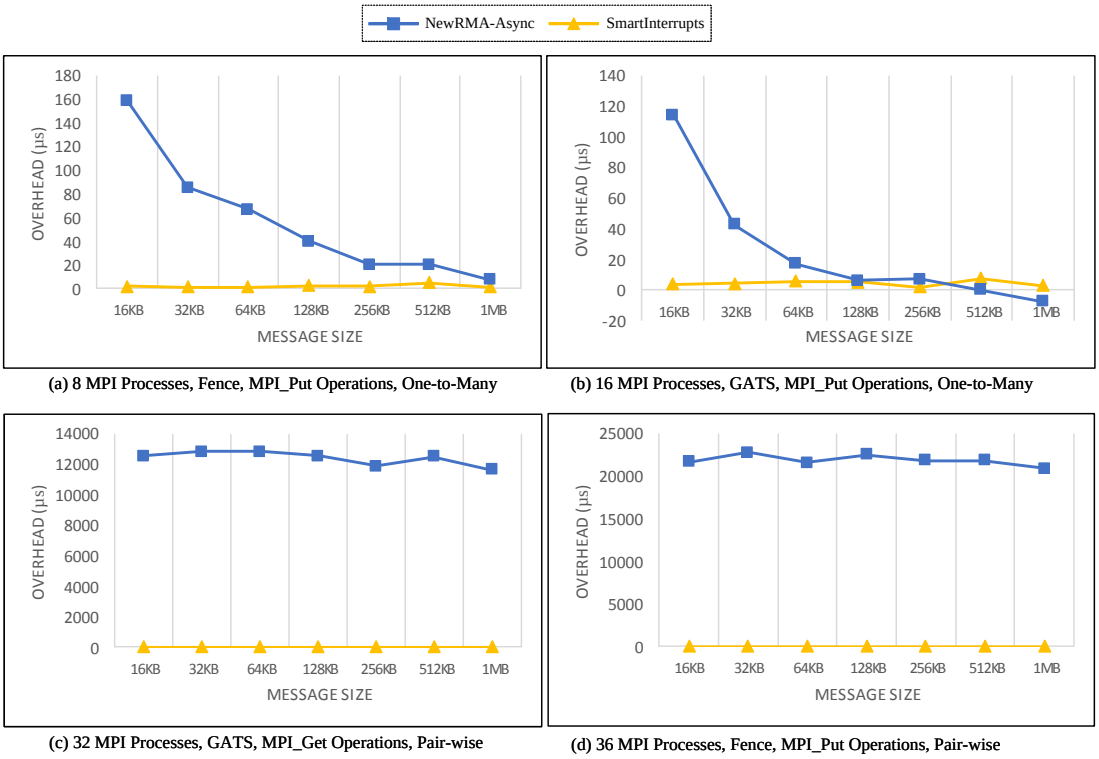


Fig. 5.6. One-Sided Latency Overhead with Fence and GATS

progression, the progress engine is always active. Therefore, the message can be progressed as soon as the control signal arrives. With 16 MPI processes per node, NewRMA-Async suffers oversubscription on at least 12 CPU cores, leading to very large overheads. In SmartInterrupts, each interrupt thread is mapped to the same core as its MPI process, but oversubscription only happens when the interrupt thread is called into action. This is possible but unlikely in the latency overhead micro-benchmarks because both the access and exposure epochs are opened at almost the same time.

Communication/Computation Overlap

Unlike two-sided communication, one-sided communication necessitates complex synchronizations among multiple peers and may involve multiple RMA messages in the same epoch. This makes it difficult to quantitatively measure the communication/computation overlap.

However, as shown in [73, 86], overlap can be well appreciated through qualitative means. The algorithm used to evaluate the overlap of SmartInterrupts is shown in Figure 5.7. A small delay is added to the target at Line 2. This delay is long enough to ensure that its exposure epoch will not open until the origin has entered its synthetic work, and prevents the initiation of any RMA communications before the synthetic work. Overlap will be confirmed if the epoch duration is less than the sum of the message latencies and the synthetic work.

ORIGIN:

```
1. MPI_Barrier(MPI_COMM_WORLD)
2. Measure Start Time
3. MPI_Win_fence/MPI_Win_start
4. Foreach( target in Targets){
5.    MPI_Put/MPI_Get
6. }
7. Synthetic Work
8. MPI_Win_fence/MPI_Win_complete
9. Measure Stop Time
```

TARGET:

```
1. MPI_Barrier(MPI_COMM_WORLD)
2. Small constant delay to delay
   the opening of the exposure
   epoch
3. Measure Start Time
4. MPI_Win_fence/MPI_Win_post
5. MPI_Win_fence/MPI_Win_wait
6. Measure Stop Time
```

Fig. 5.7. Template for Fence & GATS Communication\Computation Overlap Micro-benchmark

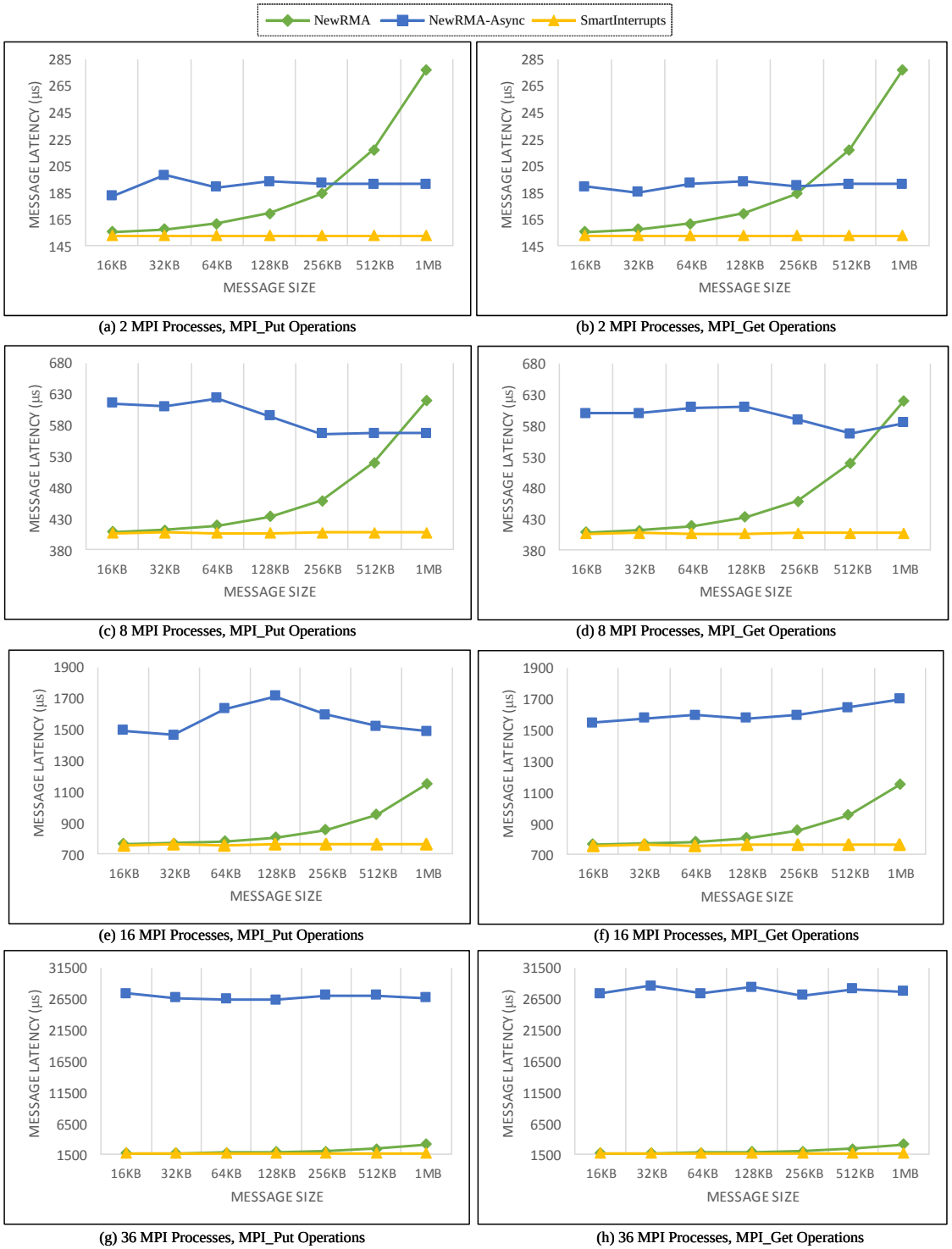
To qualitatively analyze the overlap, the algorithms described in Figure 5.7 are executed in two phases. In the first phase, the target is not delayed, that is, the delay at line 2 is zero so that the access and exposure epochs open at the same time. Also, the synthetic work is set to zero and the average epoch durations are recorded. This phase essentially measures the minimum epoch duration for a particular message size, which is dominated by the message latency. The message latency for the largest message in this phase serves as the baseline for the following phase. In the next phase, the target is delayed and the synthetic work is set to the epoch duration that was recorded for the largest message in phase one. The algorithms are then executed and the average epoch durations are recorded. Finally, the recorded results are plotted as curves in a graph. If the curve of an approach is below the baseline, then that is an indication of communication/computation overlap. An ideal curve is a straight line with a slope of zero and an intercept that is equal to the synthetic work.

Figure 5.8, Figure 5.9, Figure 5.10 and Figure 5.11 show the results of the overlap micro-benchmark with different combinations of communication schemes, MPI processes, RMA epochs and RMA operations. The only thing that is constant throughout all the results is the number of helper processes. During the experiments, it was found that one helper process was enough to achieve the ideal amount of overlap in all scenarios. In fact, the intercepts of SmartInterrupts' lines always remained in the proximity of 0.0001 percent of their ideal values. The results of NewRMA are in line with expectations. It does not offer any asynchronous progression, so the communications and computations end up being serialized. The results of NewRMA-Async are always worse than that of SmartInterrupts. It exhibits decent results for 2, 4 and 8 MPI processes, but oversubscription and constant contention for the progress engine lock makes its results uncomparable after that.

5.4.2 Exclusive Lock Micro-Benchmark

The experimental setup consisted of the same two nodes that are mentioned in Section 5.4.1. Also, parameters like the latency overhead and memory footprint are expected to be the same, as the same implementation works for both active and passive target synchronizations. However, the communication patterns and the micro-benchmark algorithms used for the analysis of fence and GATS overlap cannot be used for exclusive lock epochs. Therefore, a different micro-benchmark was developed to study the overlap in case of exclusive lock epochs. This micro-benchmark requires only three processes, two of which are designated as the origins. The algorithms executed by Origin0 and Origin1 are described in Figure 5.12 and the code at the target only consists of an MPI_Barrier. This micro-benchmark essentially simulates the scenario illustrated in Figure 5.1(c).

To analyze the overlap, the algorithms are executed in two phases. In the first phase, Origin0 is absent, so there is no need to delay Origin1 (line 2). Also, the synthetic work is set to zero and the epoch durations are recorded. This step is performed to obtain the communication latencies of



(a) 2 MPI Processes, MPI_Put Operations

(b) 2 MPI Processes, MPI_Get Operations

(c) 8 MPI Processes, MPI_Put Operations

(d) 8 MPI Processes, MPI_Get Operations

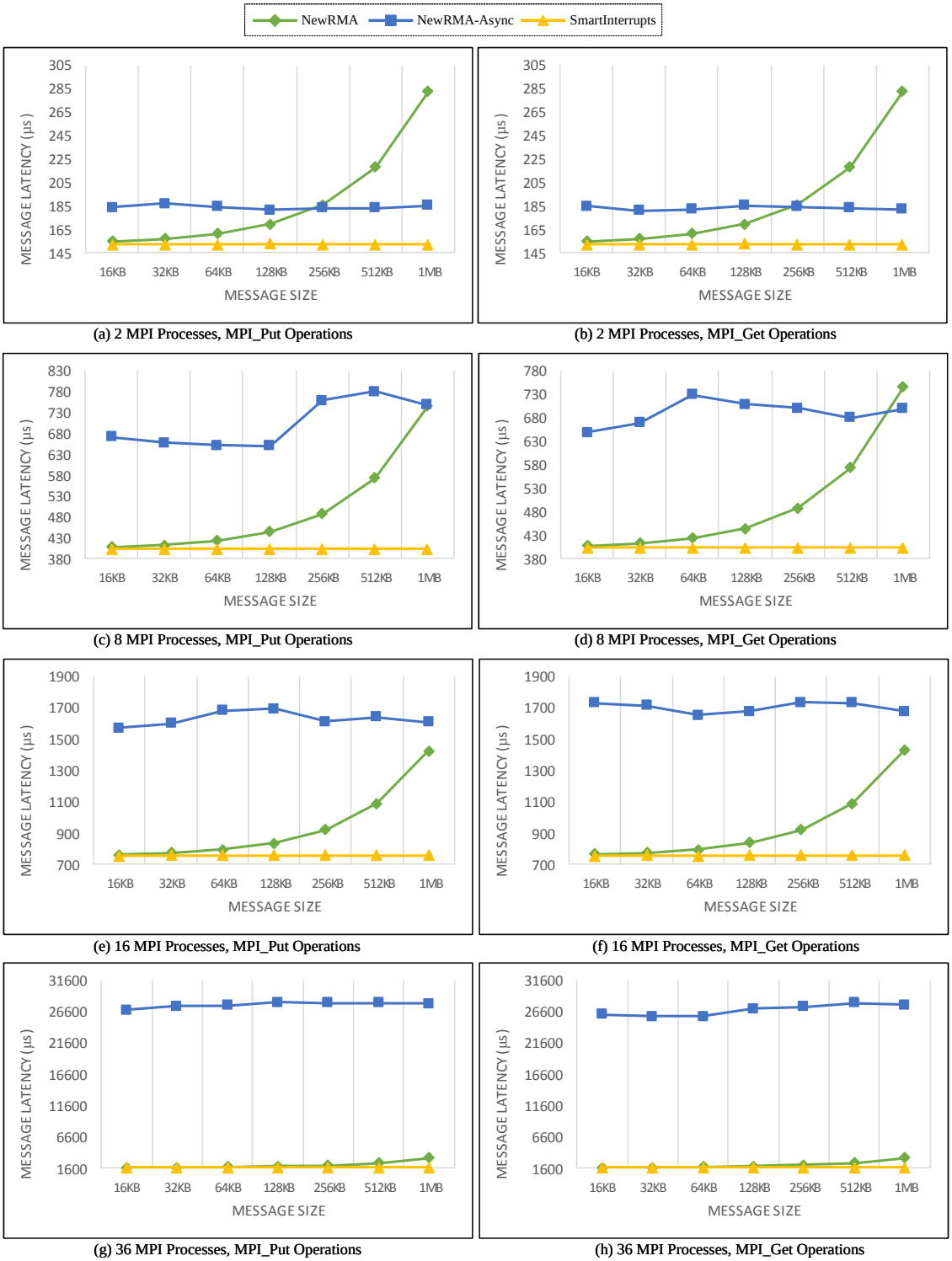
(e) 16 MPI Processes, MPI_Put Operations

(f) 16 MPI Processes, MPI_Get Operations

(g) 36 MPI Processes, MPI_Put Operations

(h) 36 MPI Processes, MPI_Get Operations

Fig. 5.8. One-Sided Overlap Results for Fence Epochs with Pair-Wise Communication Pattern and 1 HP



(a) 2 MPI Processes, MPI_Put Operations

(b) 2 MPI Processes, MPI_Get Operations

(c) 8 MPI Processes, MPI_Put Operations

(d) 8 MPI Processes, MPI_Get Operations

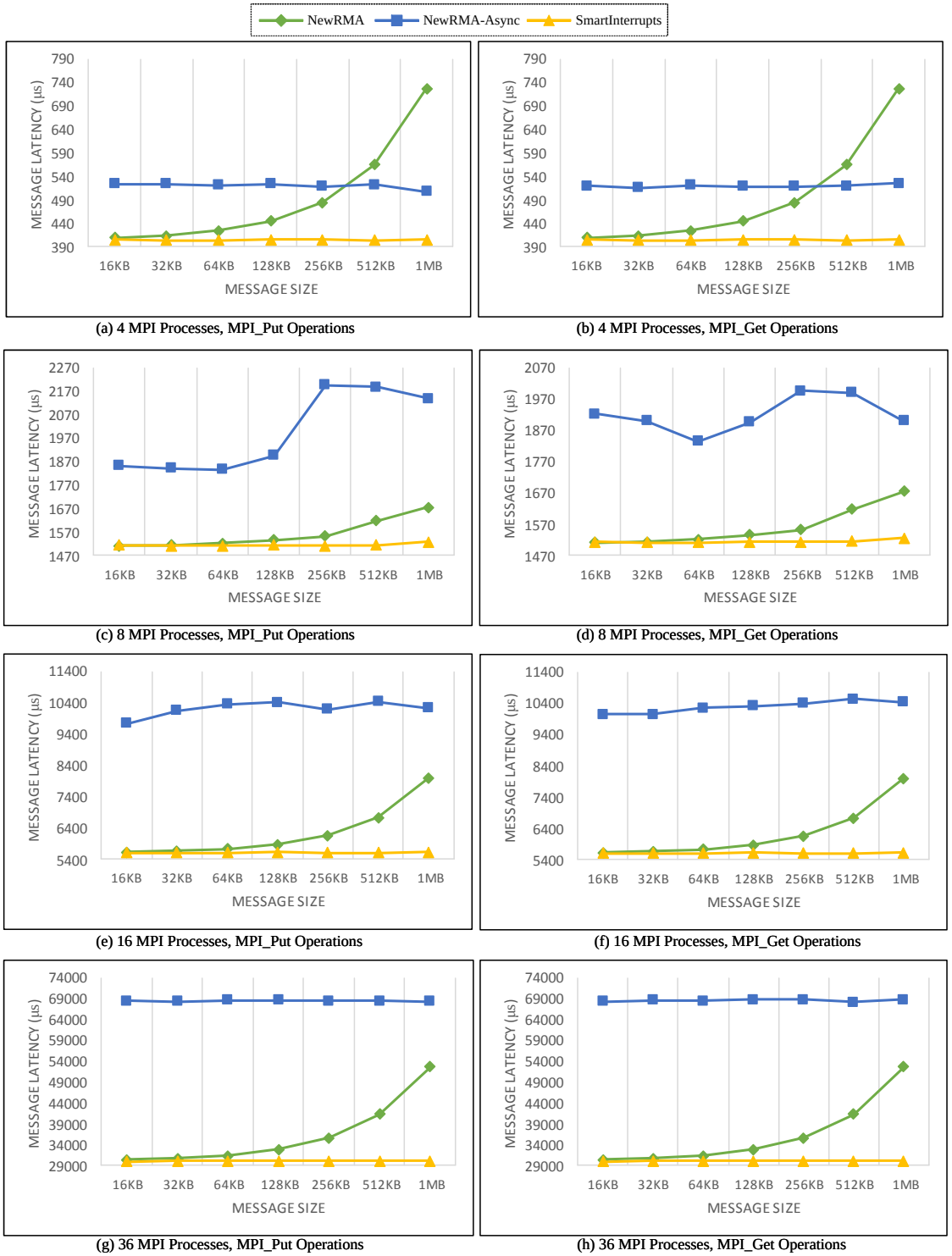
(e) 16 MPI Processes, MPI_Put Operations

(f) 16 MPI Processes, MPI_Get Operations

(g) 36 MPI Processes, MPI_Put Operations

(h) 36 MPI Processes, MPI_Get Operations

Fig. 5.9. One-Sided Overlap Results for GATS Epochs with Pair-Wise Communication Pattern and 1 HP



(a) 4 MPI Processes, MPI_Put Operations

(b) 4 MPI Processes, MPI_Get Operations

(c) 8 MPI Processes, MPI_Put Operations

(d) 8 MPI Processes, MPI_Get Operations

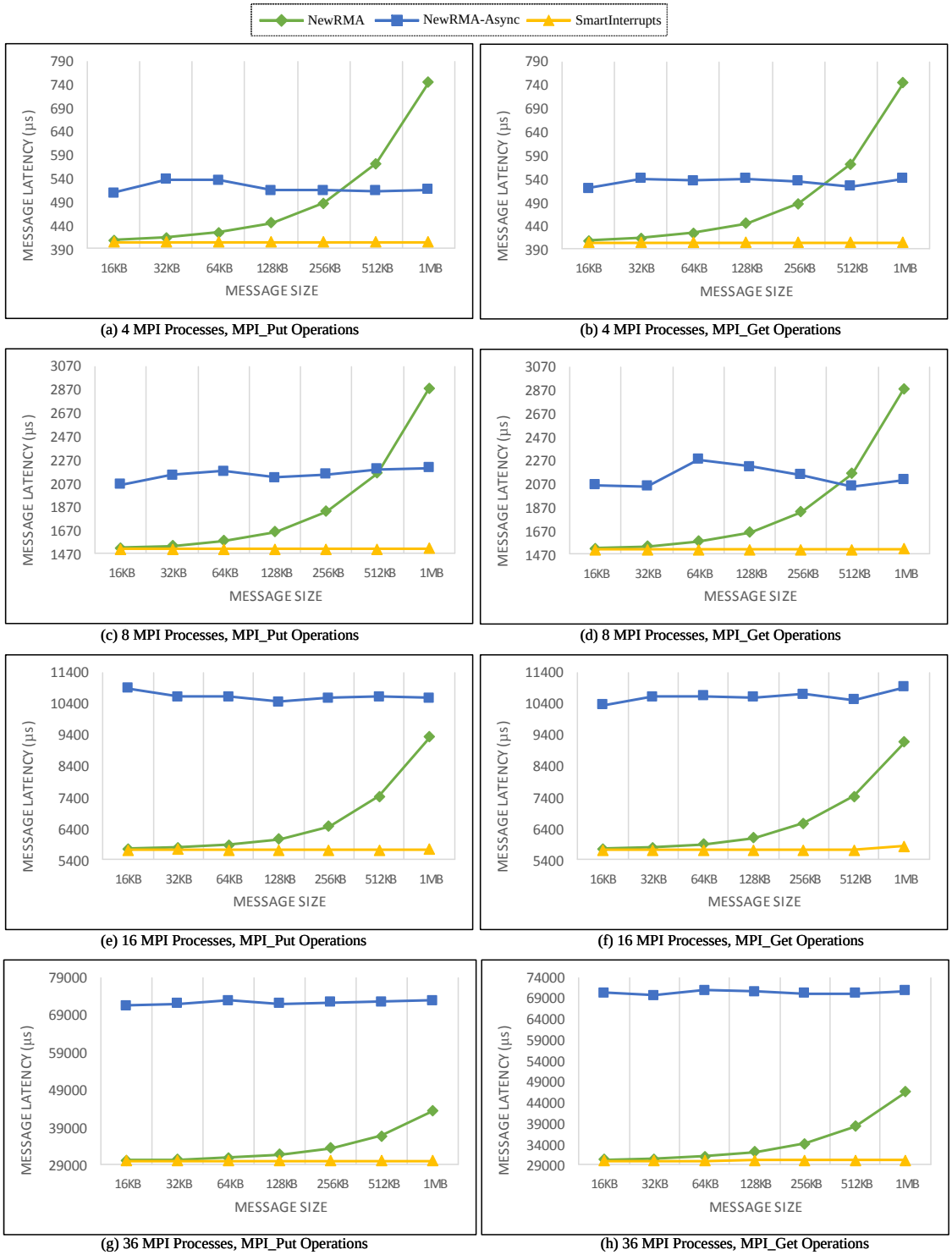
(e) 16 MPI Processes, MPI_Put Operations

(f) 16 MPI Processes, MPI_Get Operations

(g) 36 MPI Processes, MPI_Put Operations

(h) 36 MPI Processes, MPI_Get Operations

Fig. 5.10. One-Sided Overlap Results for Fence Epochs with One-to-Many Communication Pattern and 1 HP



(a) 4 MPI Processes, MPI_Put Operations

(b) 4 MPI Processes, MPI_Get Operations

(c) 8 MPI Processes, MPI_Put Operations

(d) 8 MPI Processes, MPI_Get Operations

(e) 16 MPI Processes, MPI_Put Operations

(f) 16 MPI Processes, MPI_Get Operations

(g) 36 MPI Processes, MPI_Put Operations

(h) 36 MPI Processes, MPI_Get Operations

Fig. 5.11. One-Sided Overlap Results for GATS Epochs with One-to-Many Communication Pattern and 1 HP

ORIGIN0:	ORIGIN1:
1. MPI_Barrier(MPI_COMM_WORLD) 2. MPI_Win_lock 3. Small delay to ensure that Origin1 enters the synthetic work without obtaining the lock. 4. MPI_Win_unlock 5. Measure Stop Time	1. MPI_Barrier(MPI_COMM_WORLD) 2. Small delay to ensure that Origin0 gets the lock first 3. Measure Start Time 4. MPI_Win_lock 5. MPI_Put/MPI_Get 6. Synthetic Work 7. MPI_Win_unlock 8. Measure Stop Time

Fig. 5.12. Template for Exclusive Lock Communication\Computation Overlap Micro-benchmark

messages of different size. In the next phase, Origin0 is activated and Origin1's synthetic work is set to the epoch duration that was recorded for the largest message in phase one. These algorithms were executed 1000 times and the average epoch durations of the last 900 iterations were recorded and graphed. The results are displayed in Figure 5.13. They are very similar to the Fence and GATS overlap results and can be interpreted in the same way as described in the overlap micro-benchmark section in Section 5.4.1.

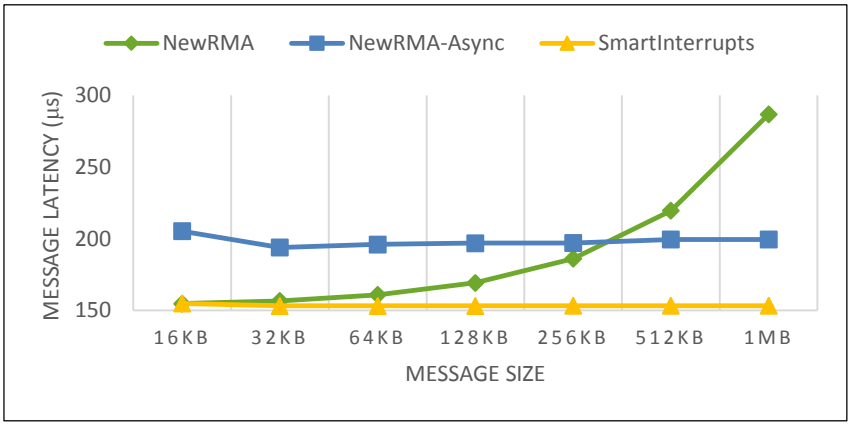


Fig. 5.13. Overlap Micro-benchmark Results for Exclusive Lock Epochs with 1 HP

5.4.3 Memory Footprint Analysis

The memory footprint analysis of SmartInterrupts' one-sided implementation was performed in the same way as described in Section 4.5.1, that is, by inspecting the VmRss field in the status

file of each MPI process. The number of MPI processes was raised incrementally to understand if the memory footprint scales accordingly. SmartInterrupts' addition to the memory footprint was found to be negligible and hovered around 60 KB. Changes in the number of MPI processes had no effect on this value.

5.4.4 Application Results

SmartInterrupts' real-world performance was assessed by testing it with the **Lower Upper (LU) decomposition** application [88]. This application factorizes the matrix as the product of a lower triangular matrix and an upper triangular matrix. Its most important purpose is to solve large systems of linear equations and is used in many scientific and engineering domains, including thermodynamics and fluid dynamics. Matrices of various dimensions can be fed to this application, with larger matrices obviously requiring more computation resources.

The application was executed with SmartInterrupts using differently-sized matrices and different number of MPI processes. Similar experiments were repeated using NewRMA and the results were compared. From the micro-benchmark results, it is evident that NewRMA-Async does not lead to any performance improvements, even without oversubscription. Also, a very few CPU cores were spared on each node, which would have led to oversubscription. These reasons guaranteed the inferior performance of NewRMA-Async. Hence, it was not used in evaluating SmartInterrupts' application performance. Figure 5.14 compares the application execution results of SmartInterrupts and NewRMA with strong scaling. For these results, the size of the matrices was constant, with dimensions of **32768x32768**. The experiments were performed using 425, 544 and 576 CPU cores, excluding one core per node for the helper process. The experimental cluster consisted of 640 CPU cores. A maximum of 576 non-oversubscribed MPI processes could be executed on this cluster after accounting for one spare core per node for the OS processes and one helper process per node. Similarly, a maximum of 544 CPU cores could be used for MPI processes

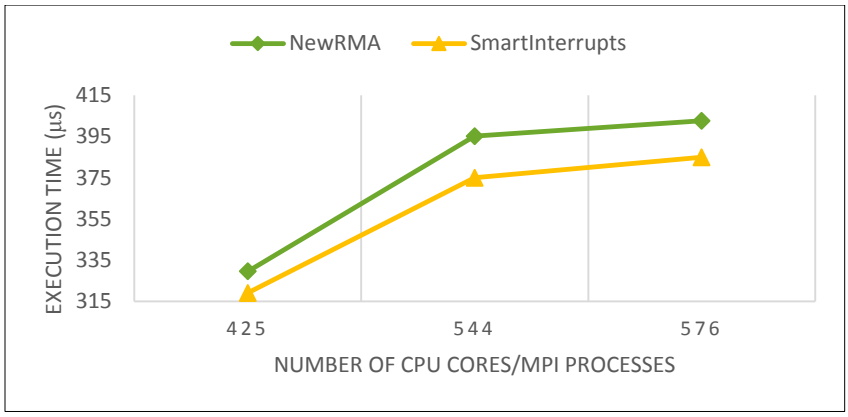


Fig. 5.14. LU Decomposition Results with 1 Helper Process per Node

if two cores per node were left spare for the OS processes with one helper process per node. These configurations were executed over 32 nodes. To compare against the performance results of 576 and 544 MPI processes, the process count of 425 MPI processes was chosen that fully occupied 25 nodes after accounting for three spare cores per node for the OS and helper processes. The graph shows consistently lower execution times for SmartInterrupts, with performance improvements of 3.1, 5.1 and 4.4 percent for 425, 544 and 576 MPI processes respectively. This suggests that SmartInterrupts can indeed make a difference in a real-world scenario.

5.5 Summary

This chapter extended the design and implementation of two-sided SmartInterrupts to one-sided communication. The principal idea behind the two designs is similar, that is, to have a few processes that assist in node-wide message progression, where the task of asynchronous progression is performed by interrupt threads. Some existing approaches propose non-blocking calls for epoch opening and closing. Non-blocking calls facilitate overlap but there are still some scenarios in which the communication and computation end up being serialized. For example, in a fence epoch, the origin may have pending RMA operations queued at the middleware and be involved in a long computation. Meanwhile, if the target opens its epoch, the origin would remain

oblivious about it until the end of its computation, causing no overlap. Asynchronous progression techniques have been proposed to address such scenarios, but are associated with their own issues. Also, there is no practical solution that works well for both two-sided and one-sided communications. SmartInterrupts for one-sided communication is implemented on top of NewRMA, in which all epoch opening and closing calls are non-blocking. It does not suffer the same drawbacks as other asynchronous progression techniques like polling and Casper, and has been shown to work well for one-sided communications.

In this design, each MPI process is augmented with an interrupt thread that progresses the RMA operations in parallel to the application thread. The interrupts to the progression thread are generated by the helper processes which receive decision making information from the MPI processes. Several helper processes can be spawned and each of them can be associated with multiple MPI processes. The number of helper processes per node can be set as an environment variable before the execution of the MPI application. Each MPI process shares three types of information with its helper process, ICD, IRD and the progress engine status. The ICD is the status of the targets' exposure epoch. If an origin does not find its exposure epoch open, it requests interrupts from its helper process through ICD. Each helper process matches the IRDs and ICDs, and triggers interrupts to the appropriate progression thread if a match is found and the progress engine lock is available.

The design was evaluated by testing it with micro-benchmarks and the LU decomposition application. The results of the tests were compared to those of NewRMA and NewRMA with polling based asynchronous progression (NewRMA-Async). Micro-benchmarks were designed to analyze the latency overhead, communication/computation overlap and memory footprint of the different approaches. The latency overhead and memory footprint of SmartInterrupts was found to be negligible. The tests with the overlap micro-benchmark showed consistently better results for SmartInterrupts than the other test subjects, and close to the ideal values. To assess the applicability

of this approach in a real-world scenario, the LU decomposition application was executed using NewRMA and SmartInterrupts. The results were in favor of SmartInterrupts, showing a consistent reduction in the application execution times. **Consequently, this chapter successfully answers the third research question posed in Section 1.2.**

Chapter 6

Conclusion and Future Work

High Performance Computing has become commonplace in several scientific, engineering and economic domains. HPC applications are time-critical and are executed on large scales; therefore, improving the execution times of HPC applications is important. Increasing the parallelization of an application generally leads to an increase in communications due to the increased need for synchronizations and to exchange intermediate results. One of the ways to improve the performance of parallel applications is by overlapping communications with computations. This is a latency hiding technique which is made possible by modern NICs that offer offload capabilities. MPI is the de-facto standard used in HPC today, and most scientific applications are written in MPI. This thesis discusses the inefficiencies associated with the rendezvous protocols in point-to-point communications and RMA synchronizations in one-sided communications, and proposes an approach to improve the communication/computation overlap in MPI applications [47].

Chapter 2 laid the foundation for the research explored in this thesis. It presented a discussion on HPC and the different messaging semantics of MPI. Chapter 3 highlighted the different scenarios that inhibit communication/computation overlaps in MPI communications, and presented a literature review on the approaches that aim to address such inefficiencies. Finally, Chapter 4 and Chapter 5 presented the design, implementation and performance evaluation of a novel

asynchronous message progression technique, SmartInterrupts, for two-sided and one-sided communications in MPI.

6.1 Summary of Findings

Chapter 4 proposed an asynchronous message progression technique called SmartInterrupts to improve the overlap of point-to-point communications. Unlike protocol improvement methods, SmartInterrupts behavior is completely deterministic, meaning that there is no randomness involved. The action will always be preceded by a predetermined set of events. Unlike, hardware-based approaches, it does not require specialized hardware and only requires a few spare CPU cores. Among host-based approaches, the most common is asynchronous message progression, which may be based on polling or interrupts. SmartInterrupts is a hybrid approach that harnesses the strengths of both polling and interrupt based approaches while avoiding their disadvantages. Two-sided SmartInterrupts uses the sender initiated RDMA Read based rendezvous protocol to leverage its natural overlap potential when the sender arrives first. In this approach, the progression is performed by interrupt threads which are triggered into action by the helper processes when they find a match between the incoming control signals and the queued interrupt requests. Each MPI process has its own interrupt thread and several interrupt threads may be associated with a single helper process.

The performance of SmartInterrupts was evaluated using two-sided and collective micro-benchmarks, as well as the NAS SP application. The results of two-sided micro-benchmarks showed that SmartInterrupts incurs negligible overhead and achieves up to 100 percent overlap in most scenarios. The design scales well with the number of MPI processes and message size, and exhibits consistently lower message latencies compared to other approaches. It adds about 300KB per process to the memory footprint, which is not a threat to scalability. Experiments with collective micro-benchmarks showed that SmartInterrupts can improve the overlap of MPI_Ialltoall;

however, other collectives are not amenable to SmartInterrupts' design. For such collectives, SmartInterrupts does not adversely impact performance. Finally, executing NAS-SP with SmartInterrupts showed a performance improvement of up to 19 percent when compared to its execution with no progression threads.

Chapter 5 extended SmartInterrupts' design to improve the overlap of one-sided communications. Similar to its two-sided counterpart, one-sided SmartInterrupts is a node-wide asynchronous message progression technique that uses both polling and interrupts. This approach enables the asynchronous progression of pending RMA messages when their control signals arrive while the origin is busy in computations. Asynchronous progression techniques have been proposed to address such scenarios, but are associated with their own issues. Also, there is no practical solution that works well for both two-sided and one-sided communications. SmartInterrupts for one-sided communication is implemented on top of NewRMA, in which all epoch opening and closing calls are non-blocking. It does not suffer the same drawbacks as other asynchronous progression techniques like polling and Casper, and has been shown to work well for one-sided communications. As with the two-sided design, the interrupts to the progression thread are generated by the helper processes which receive decision making information from the MPI processes. The helper processes are kept informed about the opening of exposure epochs and the granting of locks through shared buffers. When an MPI process does not find its control signal, it adds a request to its helper process for interrupts through a shared buffer. An interrupt is triggered by the helper process to the progression thread of this MPI process when its control signal arrives, provided that the progress engine is not already active in the main thread. Each helper processes can be associated with multiple MPI processes and the number of helper processes can be set as an environment variable before the execution of the MPI application.

The design was evaluated using micro-benchmarks and the LU Decomposition application. The results of the tests were compared to those of NewRMA and NewRMA with polling based

asynchronous progression (NewRMA-Async). The overlap micro-benchmarks with fence, GATS and exclusive lock epochs showed consistently much better results for SmartInterrupts than the other approaches. From fence and GATS micro-benchmarks it was found that one-sided SmartInterrupts contributes negligibly to the overheads and adds to the memory footprint of the application by about 60KB per process. To assess the applicability of SmartInterrupts in a practical scenario, it was tested with the LU Decomposition application, where it resulted in a modest performance improvement over NewRMA.

6.2 Future Work

6.2.1 Unifying One-Sided and Two-Sided Designs

The design of SmartInterrupts was first conceived for point-to-point communications. The choice to implement the design on MVAPICH2-2.2 was based on the fact that it was the latest version of MVAPICH2 that incorporated the latest revisions of the MPI standard and contained the most optimized progress engine. To take advantage of intra-node one-sided communication overlaps proposed in [86] and the entirely non-blocking RMA synchronizations proposed in [88], which are implemented in NewRMA, one-sided SmartInterrupts was implemented on top of NewRMA. However, NewRMA is implemented on top of MVAPICH2-1.9.

Since the core components and the design principle of point-to-point and one-sided SmartInterrupts are similar, it is possible to unify the two designs into a single implementation. This unification requires the consideration of the following components: the progress engine, the interrupt thread, the Interrupt Handler kernel module, the ICB and IRB shared buffers and the helper process. The implementations of the interrupt thread and the kernel module are exactly the same in both versions. The ICB and IRB serve the same purpose in both implementations; however,

the data in them are of different sizes and are required to be interpreted differently. This can be addressed by creating two segments in the shared buffers, one for each type of communication.

The message matching algorithm used in the polling loop of the helper process is different for the two implementations. One way to address this would be to create two polling threads at each helper process, where each thread polls dedicatedly for one of the message types. However, this solution may lead to suboptimal resource utilization. The other approach could be to serialize the two algorithms in a single polling loop. This idea can further be augmented by prioritization variables that can be controlled by the user. If an application prefers one of the messaging types then the polling for it can be prioritized over the other. The biggest challenge to unification would be the integration of the two progress engines. Both designs required the modification of the progress engine in different ways. However, the modifications are mutually exclusive and would not have any effect on the performance of the other. As mentioned previously, the progress engine for the one-sided implementation encompasses the designs of [86] and [88]. The implementation of the two-sided and one-sided designs required significant changes to their respective vanilla progress engines. However, modifications to the progress engine for two-sided SmartInterrupts were comparably fewer. Therefore, the integration of the two progress engines would require the porting of the two-sided implementation to the one-sided implementation. But, to take advantage of the most updated middleware, the integration of the designs would first require the porting of one-sided SmartInterrupts to MVAPICH2-2.2.

6.2.2 Improving the Performance of SmartInterrupts for Collectives

As mentioned in the previous section, SmartInterrupts does not improve the overlap of all the collectives. The reason for this is the scheduling algorithms that are used at the middleware to implement these collectives. These algorithms hinder the timely progression of collective messages by not supplying enough interrupt triggering information to the helper processes. During the

experimentation, the naïve scheduling algorithm for MPI_Igather was found to perform significantly better than the optimized binomial tree algorithm. Therefore, one potential avenue that could be explored in the future is to experiment with different scheduling algorithms for the non-compliant collectives.

6.2.3 Eliminating Futile Interrupts in One-Sided SmartInterrupts

As discussed in Section 5.3.2, there is one particular scenario in one-sided SmartInterrupts that can lead to the triggering of futile interrupts. This becomes possible when the MPI process is the origin for multiple targets. An undesired interrupt may be triggered if an origin has requested SmartInterrupts' support and is expecting a control message from a particular target, and in the meantime, some other target sends the control information. The current known solution to this problem consists of having a distinct representation in the ICB for the control signal of each MPI process. However, this approach is not scalable as it would require the scaling of the ICB from one to as much as the number of MPI process. Although the overhead due to this issue is not very significant, finding a solution to this could be a potential research work.

6.2.4 Dynamically Enabling/Disabling the SmartInterrupts Mechanism

As was seen in the NAS SP application results in Section 4.5.4, there may be certain scenarios where SmartInterrupts does not appreciably improve the performance of an MPI application. Although the two-sided and one-sided overhead micro-benchmarks show that SmartInterrupts does not affect the application performance adversely, it is unjustified to occupy and utilize the CPU if there is no practical outcome from it. Therefore, another future direction to this research could be to investigate an approach in which the SmartInterrupts mechanism can be enabled or disabled dynamically during runtime. While the interrupt threads are sleeping, they do not utilize the CPU, so the entire mechanism can be effectively disabled by forcing the helper processes to either exit

or sleep. The helper process is the only element of SmartInterrupts that consumes CPU cycles even when it is not contributing positively. Therefore, disabling the helper process would essentially eliminate any consumption of CPU cycles by SmartInterrupts.

6.2.5 Investigating Multi-threaded MPI Applications

Presently, the thread-safety of the two implementations has not been fully analyzed. Therefore, the performance with multi-threaded MPI applications cannot be reliably predicated at this moment. One of the known thread-safety issue is the addition of interrupt requests to the IRB. The thread-safety of SmartInterrupts will be looked at in the future.

Bibliography

[1] S. Ahuja, N. Carriero, and D. Gelernter. 1986. Linda and Friends. Computer 19, 8 (August 1986), 26-34.

[2] G. Almasi. PGAS (Partitioned Global Address Space) Languages, Encyclopedia of Parallel Computing, 2011. Springer US, Boston, MA, pp. 1539—1545

[3] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Tech. rep. DARPA IPTO, Air Force Research Laboratory, Sept. 14, 2009., D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Tech. rep. DARPA IPTO, Air Force Research Laboratory, Sept. 14, 2009.

[4] AMD HyperTransport Technology, http://www.amd.com/en-us/innovations/software-technologies/hypertransport, accessed 2016-12-12.

[5] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni and R. Rajamony. The PERCS High-Performance Interconnect. 2010 18th IEEE Symposium on High Performance Interconnects, Mountain View, CA, 2010, pp. 75-82.

[6] T. Armstrong, M. G. Burke, R. Butler, B. L. Chamberlain, S. Chandrasekaran, B. Chapman, J. Daily, J. Dinan, D. Eachempati, I. T. Foster, W. D. Gropp, P. Hargrove, W. Hwu, N. Jain, L. Kale, D. Kirk, K. Knobe, A. Krishnamoorthy, J. A. Kuehn, A. Kukanov, C. E. Leiserson, J. Lifflander, E. Lusk, T. Mattson, B. Palmer, S. C. Pieper, S. W. Poole,

A. D. Robison, F. Schlimbach, R. Thakur, A. Vishnu, J. M. Wozniak, M. Wilde, K. Yelick, Y. Zheng. Programming Models for Parallel Computing. P. Balaji (Ed.). MIT Press, 2015.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI On Millions Of Cores. Parallel Processing Letters 2011 21:01, 45-60

[8] B. Barrett, G. Shipman, and A. Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 4757 of Lecture Notes in Computer Science, pages 242–250. Springer Berlin Heidelberg, 2007.

[9] B. W. Barrett, R. Brightwell, K. S. Hemmert, K. B. Wheeler, and K. D. Underwood, Using Triggered Operations to Offload Rendezvous Messages. In Proceedings of the 18th European MPI Users' Group Meeting (EuroMPI '11), pages 120-129, 2011.

[10] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, A. Nataraj, Benchmarking the effects of operating system interference on extreme-scale parallel machines, Cluster Computing, vol. 11, March 2008.

[11] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood and R. C. Zak. Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, Santa Clara, CA, 2015, pp. 1-9.

[12] R. Brightwell, B. Barrett, K. Hemmert, and K. Underwood. Challenges for High-Performance Networking for Exascale Computing. In Proceedings of the 2010 International Conference on Computer Communications and Networks (ICCCN), pages 1–6, 2010.

[13] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, 2010, pp. 180-186.

[14] B. Chapman, G. Jost and R. Van Der Pas. Using OpenMP: portable shared memory parallel programming. Vol. 10. MIT press, 2008.

[15] Characterization of the Department Of Energy Scientific Applications, https://portal.nersc.gov/project/CAL/designforward.htm, accessed 2016-12-12.

[16] A. Danalis, A. Brown, L. Pollock, M. Swany, and J. Cavazos. Gravel: A Communication Library to Fast Path MPI. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 5205 of Lecture Notes in Computer Science, pages 111–119. Springer Berlin Heidelberg, 2008.

[17] K. Davis, A. Hoisie, G. Johnson, D. Kerbyson, M. Lang, S. Pakin, and F. Petrini. A Performance and Scalability Analysis of the BlueGene/L Architecture. In Proceedings of the 2004 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 41–, Nov 2004.

[18] A. Denis. Pioman: a Generic Framework for Asynchronous Progression and Multithreaded Communications. In 2014 IEEE International Conference on Cluster Computing (CLUSTER), Madrid, 2014, pages 276-277.

[19] A. Denis. Pioman: a Pthread-Based Multithreaded Communication Engine. In 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pages 155-162, 2015.

[20] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes and F. W. Atos. The BXI Interconnect Architecture. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, Santa Clara, CA, 2015, pp. 18-25.

[21] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. 2010. Hybrid parallel programming with MPI and unified parallel C. In Proceedings of the 7th ACM international conference on Computing frontiers (CF '10). ACM, New York, NY, USA, 177-186.

[22] D. Doerfler and R. Brightwell. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User's Group Meeting Bonn Germany, September 17-20, 2006 Proceedings. 2006, Berlin, Heidelberg, pages 331-338.

[23] T. El-Ghazawi and L. Smith. UPC: unified parallel C. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06). ACM, New York, NY, USA. Article 27.

[24] 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. High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, Salt Lake City, UT, 2012, pp. 1-9.

[25] I. Foster, W. Gropp, and C. Kesselman. 2003. Message passing and threads. In Sourcebook of parallel computing, J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 313-329.

[26] M. Gollcbiewski and J. L. Tr¨aff. MPI-2 One-Sided Communications on a Giganet SMP Cluster. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 2131 of Lecture Notes in Computer Science, pages 16–23. Springer Berlin Heidelberg, 2001.

[27] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. ConnectX-2 InfiniBand Management Queues: First Investigation of the New Support for Network Offloaded Collective Operations. In Proceedings of the 2010 IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID), pages 53–62. IEEE Computer Society, 2010.

[28] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. .Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities. In IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pages 1-8, 2010.

[29] W. Gropp. MPI: The Once and Future King. Keynote at EuroMPI16, September, 2016. Edinburgh, Scotland.

[30] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An Open Source Environment for Cell Broadband Engine System Software. Computer, 40(6):37–47, 2007.

[31] T. Gysi, J. Bär and T. Hoefler. dCUDA: Hardware Supported Overlap of Computation and Communication. In The International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’16), Salt Lake City, Utah, USA, November 2016.

[32] K. S. Hemmert, B. Barrett, and K. D. Underwood. Using triggered operations to offload collective communication operations. In Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface (EuroMPI'10). R. Keller, E. Gabriel, M. Resch, and J. Dongarra (Eds.). Springer-Verlag, Berlin, Heidelberg, pages 249-256.

[33] M.-A. Hermanns, M. Geimer, B. Mohr, and F.Wolf. Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 5759 of Lecture Notes in Computer Science, pages 31–41. Springer Berlin Heidelberg, 2009.

[34] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread?. In Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster), pages 213–222, 2008

[35] T. Hoefler, T. Schneider, and A. Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11.

[36] InfiniBand Trade Association, http://www.infinibandta.org/, accessed 2016-12-12.

[37] G. Inozemtsev and A. Afsahi. Designing an Offloaded Nonblocking MPI Allgather Collective Using CORE-Direct. In Proceedings of the 2012 IEEE International Conference on Cluster Computing (Cluster), pages 477–485, 2012.

[38] Intel QuickPath Interconnect, http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, accessed 2016-12-12.

[39] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In Euro PVM/MPI, ser. Lecture Notes in Computer Science, 2004, vol. 3241, pp. 68–76.

[40] W. Jiang, J. Liu, H. Jin, D. K. Panda, W. Gropp and R. Thakur. High performance MPI-2 one-sided communication over InfiniBand. Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on, 2004, pp. 531-538.

[41] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, TreadMarks: distributed shared memory on standard workstations and operating systems. In Proceedings of the USENIX

Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94). USENIX Association, Berkele

[42] M. J. Koop, A. P. Sampat and D. K. Panda. Veloblock: Efficient and Scalable RDMA Fast Path for InfiniBand. OSU-CISRC-5/09-TR20.

[43] M. Koop, J. Sridhar, and D. Panda. TupleQ: Fully-Asynchronous and Zero-Copy MPI over InfiniBand. In Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pages 1–8, 2009.

[44] M. Koop, J. Sridhar, and D. Panda. Scalable MPI Design over InfiniBand Using eXtended Reliable Connection. In Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster), pages 203–212, 2008.

[45] A. Kühnal, M.-A. Hermanns, B. Mohr, and F.Wolf. Specification of Inefficiency Patterns for MPI-2 One-Sided Communication. In Proceedings of the 2006 Euro-Par Conference, volume 4128 of Lecture Notes in Computer Science, pages 47–62. Springer, August - September 2006.

[46] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman and D. K. Panda. Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 5205 of Lecture Notes in Computer Science, pages 185–193. Springer Berlin Heidelberg, 2008.

[47] K. Kumar, J. A. Zounmevo and A. Afsahi. Smart Interrupts: A Node-Wide Asynchronous Message Progression Technique. To be submitted.

[48] B. Lewis and D. J. Berg. Multithreaded Programming with Pthreads. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 1998.

[49] 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 the 2004 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2004.

[50] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. 2014. Experiences at scale with PGAS versions of a Hydrodynamics application. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14). ACM, New York, NY, USA, , Article 9 ,

[51] A. Marathe, D. Lowenthal, Z. Gu, M. Small, and X. Yuan. Profile Guided MPI Protocol Selection for Point-to-Point Communication Calls. In Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pages 733–739, 2011.

[52] Mellanox Technologies, https://www.mellanox.com/, accessed 2016-12-12.

[53] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. 2002. Critical power slope: understanding the runtime effects of frequency scaling. In Proceedings of the 16th international conference on Supercomputing (ICS '02). ACM, New York, NY, USA, 35-44.

[54] MPI Forum, http://www.mpi-forum.org/, accessed 2016-12-12.

[55] MPICH, https://www.mpich.org/, accessed 2016-12-12.

[56] MVAPICH, http://mvapich.cse.ohio-state.edu/, accessed 2016-12-12.

[57] NAS Parallel Benchmarks (NPB), https://www.nas.nasa.gov/publications/npb.html, accessed 2016-12-12.

[58] R. W. Numrich and J. Reid. 1998. Co-array Fortran for parallel programming. SIGPLAN Fortran Forum 17, 2 (August 1998), 1-31.

[59] OpenFabrics Alliance, https://www.openfabrics.org/, accessed 2016-12-12.

[60] OpenMPI, https://www.open-mpi.org/, accessed 2016-12-12.

[61] S. Pakin. Receiver-initiated Message Passing over RDMA Networks. In Proceedings of the 2008 IEEE International Parallel Distributed Processing Symposium (IPDPS), pages 1–12, 2008.

[62] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. Micro, IEEE, 22(1):46–57, 2002.

[63] Portals 4, http://www.cs.sandia.gov/Portals/portals4.html, accessed 2016-12-12.

[64] R. Rabenseifner, G. Hager and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar, 2009, pp. 427-436.doi: 10.1109/PDP.2009.43

[65] M. J. Rashti and A. Afsahi. Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects. In 15th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects 2007), Palo Alto, California, USA, August 22-24, 2007, pages 117-124.

[66] M. J. Rashti and A. Afsahi. A Speculative and Adaptive MPI Rendezvous Protocol Over RDMA-enabled Interconnects. International Journal of Parallel Programming, 37(2):223–246, 2009.

[67] M. J. Rashti and A. Afsahi. Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects. In Proceedings of the 2012 International Symposium on High Performance Computing Systems and Applications (HPCS), pages 95–101, 2008.

[68] RDMA Consortium, http://www.rdmaconsortium.org/, accessed 2016-12-12.

[69] J. Reinders, J. Jeffers, and A. Sodani. Intel Xeon Phi Processor High Performance Programming Knights Landing Edition. Elsevier, 2016.

[70] G. Santhanaraman, S. Narravula, and D. K. Panda. Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap. In 2008 IEEE International Symposium on Parallel and Distributed Processing, Miami, FL, 2008, pp. 1-11.

[71] G. Santhanaraman, T. Gangadharappa, S. Narravula, A. Mamidala, and D. Panda. Design Alternatives for Implementing Fence Synchronization in MPI-2 One-sided Communication for InfiniBand Clusters. In Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops (Cluster), pages 1–9, 2009.

[72] T. Schneider, T. Hoefler, R. E. Grant, B. Barrett and R. Brightwell. Protocols for Fully Offloaded Collective Operations on Accelerated Network Adapters. In 42nd International Conference on Parallel Processing, pages 593-602, 2013.

[73] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi and Y. Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS ’15), Hyderabad, India, May 2015, pages 665-676.

[74] M. Small and X. Yuan. Maximizing MPI Point-to-point Communication Performance on RDMA-enabled Clusters with Customized Protocols. In Proceedings of the 2009 International Conference on Supercomputing (ICS), pages 306–315. ACM, 2009.

[75] M. Small, Z. Gu, and X. Yuan. Near-Optimal Rendezvous Protocols for RDMA-Enabled Clusters. In Proceedings of the 2010 International Conference on Parallel Processing (ICPP), pages 644–652, 2010.

[76] S. Sur, H. Jin, L. Chai and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In Proceedings of the 2006 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 32–39. ACM, 2006.

[77] R. Thakur, R. Rabenseifner, and W. Gropp. 2005. Optimization of Collective Communication Operations in MPICH. Int. J. High Perform. Comput. Appl. 19, 1 (February 2005), 49-66.

[78] R. Thakur, W. Gropp, and B. Toonen. Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. International Journal of High Performance Computing Application (IJHPCA), 19:119–128, 2005.

[79] V. Tipparaju, J. Nieplocha, and D. K. Panda. 2003. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03). IEEE Computer Society, Washington, DC, USA, 84.1-.

[80] Top500, https://www.top500.org/, accessed 2016-12-12.

[81] J. L. Träff, H. Ritzdorf, and R. Hempel. The Implementation of MPI-2 One-sided Communication for the NEC SX-5. In Proceedings of the 2000 International Conference for High Performance Computing, Networking, Storage and Analysis (SC),Washington, DC, USA, 2000. IEEE Computer Society.

[82] F. Trahay, ´E. Brunet, and A. Denis. An Analysis of the Impact of Multi-Threading on Communication Performance. In 9th Workshop on Communication Architecture for Clusters (CAC), May 2009.

[83] K. Vaidyanathan, K. Pamnany, D. D. Kalamkar, A. Heinecke, M. Smelyanskiy, J. Park, D. Kim, A. Shet, G, B. Kaul, B. Joo, and P. Dubey. Improving Communication

Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters. In Proceedings of the 2014 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2014.

[84] A. Wagner, H.-W. Jin, D. Panda, and R. Riesen. NIC-based Offload of Dynamic User-defined Modules for Myrinet Clusters. In IEEE International Conference on Cluster Computing, pages 205-214, 2004.

[85] M. Wittmann, G. Hager, T. Zeiser and G. Wellein. Asynchronous MPI for the Masses. (2013) arXiv:1302.4280.

[86] J. A. Zounmevo and A. Afsahi. Intra-Epoch Message Scheduling to Exploit Unused or Residual Overlapping Potential. In Euro MPI, 2014.

[87] J. A. Zounmevo and A. Afsahi. Investigating Scenario-conscious Asynchronous Rendezvous over RDMA. In Proceedings of the 2011 IEEE International Conference on Cluster Computing (Cluster), pages 542–546, 2011.

[88] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi, Nonblocking Epochs in MPI one-sided Communication. 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing 2014), New Orleans, LA, USA, November 16-21, 2014.