Scalability-Driven Approaches to Key Aspects of the Message Passing Interface for Next Generation Supercomputing

by

Ayi Judicael Zounmevo

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
May 2014

Abstract

The Message Passing Interface (MPI), which dominates the supercomputing programming environment, is used to orchestrate and fulfill communication in High Performance Computing (HPC). How far HPC programs can scale depends in large part on the ability to achieve fast communication; and to overlap communication with computation or communication with communication.

This dissertation proposes a new asynchronous solution to the nonblocking Rendezvous protocol used between pairs of processes to transfer large payloads. On top of enforcing communication/computation overlapping in a comprehensive way, the proposal trumps existing network device-agnostic asynchronous solutions by being memory-scalable and by avoiding brute force strategies.

Achieving overlapping between communication and computation is important; but each communication is also expected to generate minimal latency. In that respect, the processing of the queues meant to hold messages pending reception inside the MPI middleware is expected to be fast. Currently though, that processing slows down when program scales grow. This research presents a novel scalability-driven message queue whose processing skips altogether large portions of queue items that are deterministically guaranteed to lead to unfruitful searches. For having little sensitivity to program sizes, the proposed message queue maintains a very good performance, on top of displaying a low and flattening memory footprint growth pattern.

Due to the blocking nature of its required synchronizations, the one-sided communication model of MPI creates both communication/computation and communication/communication serializations. This research fixes these issues and latency-related inefficiencies documented for MPI one-sided communications by proposing completely nonblocking and non-serializing versions for those synchronizations. The improvements, meant for consideration in a future MPI standard, also allow new classes of programs to be more efficiently expressed in MPI.

Finally, a persistent distributed service is designed over MPI to show its impacts at large scales beyond communication-only activities. MPI is analyzed in situations of resource exhaustion, partial failure and heavy use of internal objects for communicating and non-communicating routines. Important scalability issues are revealed and solution approaches are put forth.

Acknowledgments

Thanks to God for past and current graces, for the wisdom he offers for free; and for watching over me and my future.

I am deeply indebted to my supervisor, Dr. Ahmad Afsahi, for his unfailing support and for constantly pushing me toward the completion of this research. My thanks go to him for his guidance for high quality research, understanding and praiseworthy patience. I am also indebted to Dr. Dries Kimpe, Dr. Pavan Balaji and Dr. Rob Ross of the U.S. Argonne National Laboratory, IL. I undeniably grew with their advice and insight during my Ph.D. studies. To the members of my thesis committee, Dr. Ying Zou, Dr. Ahmed Hassan, Dr. Michael Korenberg and Dr. Jesper Larsson Träff, thanks for the priceless feedback on this research.

To my co-workers from the Parallel Processing Research lab, Dr. Ying Qian, Dr. Mohammad Rashti, Dr. Ryan Grant, Dr. Reza Zamani, Grigory Inozemtsev, Iman Faraji and Hessam Mirsadeghi, thanks for your support, the constructive discussions, the jokes and the laughters. The experience would have been lonely without you. To Ryan, additional thanks for those spontaneous and extremely helpful advice in my last year. To Iman and Hessam, thanks for now being two irreplaceable friends for life. Thanks to Xin Zhao of the University of Illinois Urbana Champaign for the collaboration and insightful discussions on MPI one-sided communications.

Thanks to the Natural Science and Engineering Research Council of Canada (NSERC) for supporting this research through grants to my supervisor. Thanks to the Electrical and Computer Engineering (ECE) Department of Queen's University as well as the school of graduate studies for the financial support. Thanks to the IT service and office staff of the ECE

Department of Queen's University. Special thanks to Ms. Debra Fraser and Ms. Kendra Pople-Easton for being so nice, so supportive and so available for me. My gratitude also goes to Pak Lui and the HPC Advisory Council for the supercomputing resources. Thanks to the Argonne Laboratory Computing Research Center for the supercomputing resources.

To all my friends in Montreal, I cannot acknowledge enough your support. In alphabetical order, Narcisse Adjalla, Santos Adjalla, Sami Abike Yacoubou-Chabi Yo, Sani Chabi Yo, Florent Kpodjedo, Ablan Joyce Nouaho and Fatou Tao, thank you. As for Florent, my improvised psychologist in difficult and lonely moments, additional thanks; you actually watched over me.

Lastly, I would like to acknowledge my family for always believing in me. My gratitude goes to my brothers and sisters, Ida, Cyr, Hervé, Eric, Lydia, Annick and Laure for their support and prayers. To my father Fabien and my mother Jeanne, you are the absolute mental strength that I carry with myself; I owe you so much.

Table of Contents

List of Tables

List of Figures

List of Equations

List of Code Listings

Acronyms

AEF Application Execution Flow. 45

API Application Programming Interface. 14

ARMCI Aggregate Remote Memory Copy Interface. 5

ASL Average Search Length. 111

CPU Central Processing Unit. 1

CQ Completion Queue. 28

CQE Completion Queue Entry. 28

CTS Clear To Send. 34

DDR Double Data Rate. 28

EDR Enhanced Data Rate. 28

FDR Fourteen Data Rate. 28

FLOPS Floating Point Operations Per Second. 2

GATS General Active Target Synchronization. 20

HCA Host Channel Adapter. 28

HPC High Performance Computing. 1

IBA InfiniBand Architecture. 28

IBTA InfiniBand Trade Association. 28

IETF Internet Engineering Task Force. 34

MASL Max Average Search Length. 111

MPI Message Passing Interface. 2

MQI Message Queue Item. 60

MQL	Max Queue Length. 111
MTBF	Mean Time Between Failures. 7
MX	Myrinet Express. 30
NAS	NASA Advanced Supercomputing. 53
NIC	Network Interface Card. 26
OFED	OpenFabrics Enterprise Distribution. 30
OS	Operating System. 13
PAMI	Parallel Active Messaging Interface. 29
PCIe	Peripheral Component Interconnect Express. 27
PEF	Parasite Execution Flow. 45
PGAS	Partitioned Global Address Space. 2
POSIX	Portable Operating System Interface. 47
PRQ	Posted Receive Queue. 60
QDR	Quad Data Rate. 28
QP	Queue Pair. 28
RDMA	Remote Direct Memory Access. 26
RMA	Remote Memory Access. 3
RPC	Remote Procedure Call. 189
RTR	Ready To Receive. 36
RTS	Ready To Send. 34
SDR	Single Data Rate. 28
SRQ	Shared Receive Queue. 29
TCP	Transmission Control Protocol. 26
uGNI	user Generic Network Interface. 30
UMQ	Unexpected Message Queue. 60
WQE	Work Queue Entry. 28

Chapter 1

Introduction

High Performance Computing (HPC) or *supercomputing* is the use of computing systems of large computational power to solve complex problems. HPC aims for speed of computation. Among other things, supercomputers are used for completing in a few hours computing activities that would take decades or centuries on mere workstations. It was a time where supercomputers were reserved only for governments, powerful corporations and select academic institutions. Back then, the immediate benefits of advances in HPC were reserved for a tiny fraction of the scientific community. Recently, however, HPC has evolved from fulfilling the sole needs of genomics, physics, ocean or astronomy research to become a powerful tool for just about any scientific research field and for industry as well. Psychologists are relying on supercomputers for knowledge retrieval and translation. Supercomputers are also routinely used for high frequency stock trading. More generally, any individual who is concerned about the next cancer cure, daily weather forecasting or fraud detection by credit card companies, etc., is already depending on HPC at unprecedented levels. Supercomputing has become a fast-paced research field that propells science, engineering, economy or even entertainment.

Like 84.6% of the 500 most powerful systems as of November 2013, the typical supercomputer is a cluster [106]; that is, a distributed system made of multiple compute *nodes* linked in a high speed *network*. Each node is usually a standalone multicore compute element. An HPC program, usually called *job* at runtime, is made of multiple processes distributed over the Central Processing Unit (CPU) cores of the cluster. The distributed nature of clusters establishes *communication* as one of their key aspects. Communication is a collaboration mechanism which allows the tasks in an HPC job to frequently exchange data or intermediate results with their peers. Communication is actually the single element that determines performance when computation must cross node boundaries. As a consequence, the efficiency of the communication subsystem of clusters tends to get a lot of attention in supercomputing research. Armed with decades of experience, and with performance at the core of its concerns, the HPC community has maintained programming paradigms, referred to as *message passing*, which allow explicit communication between the numerous CPU cores of clusters. The most widespread embodiment of these paradigms is the Message Passing Interface (MPI) [68]. MPI, which is a standardized paradigm whose specification is maintained by the MPI Forum [68], is the default programming tool delivered with nowadays' top supercomputers. MPI is also believed to remain a key player in supercomputing [9, 33]. In fact, for being so widely supported on existing supercomputers and for being the implementation vehicle of the massive amount of existing HPC code base, MPI remains a support even for newly emerging programming paradigms [9, 64, 98] such as the Partitioned Global Address Space (PGAS) languages [114]. Justifiably, MPI is a place to bring major and immediately impactful software-level improvements in supercomputing.

The performance of a supercomputer is measured in Floating Point Operations Per Second (FLOPS); and a major goal of HPC research is to squeeze out larger and larger amounts of FLOPS from supercomputers. With the large number of CPU cores found in supercomputers, *parallelism* becomes the means of creating performance. However, parallelism tends to suffer a diminushing return effect. For a given job, the 10000^{th} CPU core does not necessarily add the same number of FLOPS as the first and the second processors. This concern is just one of the many issues that are considered *scalability*-related. Scalability is concerned with how to create sustained performance growth via hardware and software evolution. Scalability is also concerned with how to get existing and new HPC jobs to leverage the performance growth.

Nowadays top supercomputers [106], such as Tianhe-2, Titan-Cray XK7 or Sequoia, are *petaflops*-capable machines; that is, their performance can reach the order of 10^{15} FLOPS. Nevertheless, just like nowadays' supercomputers allow feats that were impossible a decade ago, there is more to accomplish that the impressive 33.86 petaflops of Tianhe-2 are still inadequate for. Justifiably, the HPC community is seeking to push the scales even further; with the goal of reaching an exaflop (10^{18} FLOPS) by the end of the current decade. MPI and its various implementations have a key mandate in reaching the exascale goal; and for that purpose, their scalability effort must become less lenient on some seemingly benign serialization issues. A comprehensive scalability approach must also emcompass non communication-related aspects of HPC jobs at large scales. Examples of non communication-related concerns are resiliency and memory consumption not linked to data transfer. This dissertation focuses on specific changes required for MPI to successfully fulfill its partition of the challenges inherent to creating sustainable performance growth on nowadays' and future system scales.

1.1 Motivation

Any MPI activity that can generate and propagate latency is a potential scalability issue. Any MPI activity whose resource consumption pattern is linear or superlinear is a potential scalability issue as well. Latency propagation can create a snowball effect whose overall impact depends on how many processes are involved. In general, every MPI communication can have impacts beyond the sole processes that it involves. A single delay in any communication can transitively impact all the other processes of a job by propagating to subsequent communications or computations. As scales grow, latency propagation tends to become more and more harmful.

MPI offers three programming models. They are *two-sided communications*, also called *point-to-point communications*; *one-sided communications*, also called *Remote Memory Access (RMA)*; and *collective communications*. These three communication models are defined in depth in Chapter 2; but as a quick introduction, a two-sided communication involves a sender and receiver both issuing a communication call. The parameters of both calls must match for the send operation to be consumed at the receiver side. One-sided operations occur between two processes. However, unlike two-sided operations, they do not have any concept of reception. Conceptually, only one process does the communication. The one-sided initiator can remotely load or store data into another peer; sometimes unbeknownst to the process which owns the memory region manipulated by the initiator. Collective communications involve multiple peers. A collective communication either allows a process to send or receive from all the peers in a certain group of processes, or allows all the processes in a given group to simultaneously send and receive from one another.

The MPI one-sided communication is an example of major MPI feature that inherently bears serialization and latency propagation in its very specification. MPI RMA occurs inside *epochs* which are critical section-like regions enclosed by synchronizations calls. The synchronization calls that close the epochs are blocking; and because their execution can involve internal communication between multiple peers, they can generate and propagate latency; on top of explicitly creating communication/communication or communication/computation serialization.

For more than 15 years, the one-sided communication model of MPI has failed to generate enthusiasm and adoption. Nevertheless, the adoption level of MPI as a whole has never ceased to increase; meaning that MPI has done pretty well even without an accepted one-sided communication model. So, why would it matter now to worry about the MPI one-sided communication model? First, one-sided communications in general are supposed to be less subject to latency propagation than two-sided communications. Since only the initiator is involved in the data transfer, one-sided communications are decoupled from the availability of the receiving end. As a result, they offer more potential for avoiding initiator-generated latency from delaying the remote peer; and vice-versa. This reduced level of interaction is scalability-friendly. As proof of one-sided communication adequacy for nowadays and future HPC concerns, one can notice its strong prevalence in modern supercomputing network technologies. In fact, modern network technologies [6, 44, 54, 57] put forth prominent one-sided communication features. Some of those modern interconnection technologies even offer only one-sided models of communication at the core of their operating modes. InfiniBand [44], which is the most prevalent networking family in the 500 most powerful supercomputers [106] is one such example. Similarly, the Portals networking API [6], which is used by the Cray supercomputers, is built around one-sided semantics. The emerging PGAS family of HPC programming languages [114] are also built over one-sided communication frameworks such as the Aggregate Remote Memory Copy Interface (ARMCI) [105] and GASNet [99]. Furthermore, new classes of programs which perform unstructured communications [115] are extremely challenging to realize efficiently in situations where each sent message must be matched by a receive. In these situations, one-sided data transfers become the viable communication model; opening up the path for a whole set of dynamic, loosely synchronous patterns of computations.

The one-sided communication model is not an alternative to two-sided communication; it brings its own features. As virtue of MPI being the most portable and the default programming framework uniformly delivered with new supercomputers, MPI RMA is the most accessible one-sided programming model available. There is an important advantage to portability in supercomputing because of the disparity in system architectures [106, 120]. In particular, there is an advantage to being able to run the same cancer research HPC code on Tianhe-1, Tianhe-2, the Japanese K computer, any Blue Gene system or any Cray architecture; and even on future unknown supercomputer architectures [120]. Currently, such a source-level portability guarantee is the exclusivity of MPI. As a result, MPI is the place to truly leverage the many positive characteristics of one-sided communications for scalable HPC and the support of emerging communication patterns. The very specification of MPI one-sided communications needs a scalability-driven improvement .

Sometimes, latency propagation and serialization issues are not the consequence of how the communication is specified in the MPI standard; they are peculiar to the concrete realization of the specification. It is customary in MPI to enforce parallelism between communication and computation via nonblocking communication routines. A nonblocking communication is broken into the initiation phase which is nonblocking and the completion phase which must block until the data transfer completes. The initiation phase, which corresponds to the actual communication routine call, could potentially exit before the data transfer completes. Then the data transfer is expected to occur in parallel with any computation that is inserted between the two phases. The resulting parallelization between the data transfer and the inserted computation is called *communication/computation overlapping.* Communication/computation overlapping is meant for mitigating the communication latency. Thus, for a communication that lasts C_m and a computation that lasts C_p, the overall duration of both activities is the maximum of C_m and C_p instead of the sum of both as it is the case when serialization occurs. However, in many cases, even when communication/computation is expressed with the means provided by MPI at application level, the data transfer is simply internally deferred to the completion phase and occurs after the communication; leading to the very serialization that the specified behaviour was intended to avoid. In particular, when the payload of a two-sided communication is beyond a certain size, a handshake-based protocol called *Rendezvous* is used to negotiate buffer availability before the data transfer occurs. In nonblocking communications, the Rendezvous protocol can fail to achieve communication/computation overlapping. Several solutions put forth various designs of the protocol but they miss at least one scenario where the workarounds that they propose fail to be applicable [16, 65, 80, 85, 95, 96]. Another category of solutions fixes the problem in a comprehensive way but the very designs used by these solutions bear scalability issues because of their resource consumption patterns [56, 102]. The problem in this case resides in coming up with an adequate design of the Rendezvous protocol that is scalable and that fulfills two-sided communications without serialization.

Another major scalability source in MPI resides in its message queues. Message queues are not particularly linked to any specified feature of MPI. However they are internally very involved in MPI communications fulfilment. The impossibility for a process to consume in a timely fashion all the messages sent by the numerous peers running on other CPU cores inside the same or on remote nodes is just one of the many reasons why messages must be queued at receiving ends. Message queues are actually not specific to MPI; they are a fundamental mechanism used to work around the limited nature of resources such as memory buffers and other objects required for message reception and processing [6]. Message queues are used so frequently in MPI that they have been qualified as its most crucial data structures [49]. In fact, the performance of certain communication-intensive HPC jobs is simply determined by the underlying MPI message queue processing. Long message queue processing accounted for up to 60% of the communication latency in tests presented in [109]. It has been observed that MPI message queues grow with job sizes and scales [12, 13, 14, 49]. With the current and upcoming system scales, it has become imperative to approach MPI message queues in ways that account for the massive amount of peers each process can potentially communicate with. In particular, a modern and scalability-conscious message queue must not only remain reasonably fast at extreme scales, but it must account for the amount of memory resource that each CPU core can afford to reserve for an exponentially growing number of remote CPU cores to interact with.

Beyond all these specific and communication related issues that could be a hindrance to scalability, one must also be concerned about all the non-communicating aspects of MPI. Systems running with MPI are delivering quadrillions of FLOPS nowadays. As the HPC community is planning for the *exascale* by the end of the current decade, there is consensus among authors that profound architectural and programming behaviour shifts will be required [22]. MPI has gone through many major scalability-induced shifts in the past; but there are challenges that have never been encountered in any of the previous shifts. An example of one such challenges is the reduced Mean Time Between Failures (MTBF) issue that is anticipated at exascale [22, 50]. What happens if on average large MPI-based HPC jobs cannot run to completion before the next failure occurs? Would those jobs then become impossible to run to completion on average? Another issue is resource exhaustion management. How does MPI impact the HPC application when it runs out of internal memory? In fact, the scalability challenges of MPI are not limited to how to reach larger scales; they also include what happens to HPC programs once they can run at those large scales. Certain corner cases of past and nowadays scales will undoubtedly become normal operating conditions and will have to be dealt with, not as exceptional situations, but as regular HPC program execution events. Those situations are tagged “corner cases” because they are rare, difficult to reproduce and occur only in extreme conditions. For MPI to remain viable at extreme scales, those corner cases must nevertheless be revealed ahead of reaching exascale and provisioned for.

1.2 Problem Statement

The hardware manufacturers are realizing the amazing feat of periodically releasing supercomputers with faster network fabrics and larger CPU core counts. To the question of how to run an HPC program on those powerful and massive systems, the answer largely lies in MPI. MPI is undoubtedly changing along with the systems it runs on; but how timely or complete are those changes for the growing concern that is scalability? At a finer granularity, this dissertation answers the following questions:

1. How to approach the Rendezvous protocol of data transfer in a comprehensive way and without introducing new scalability issues?

2. For being a central aspect of MPI middleware, how to make sure that MPI message queues remain fast when job sizes grow? What data structure design approach can fulfill that speed goal without exhibiting an impractical pattern of memory consumption?

3. How does MPI evolves its one-sided communication model to make it viable and attractive for HPC program use? In particular how to solve the issue of blocking synchronization and remove the resulting serialization and latency propagation without introducing consistency hazards in the memory regions touched by one-sided communications?

4. Beyond its obvious duty of transferring data between the processes of an HPC job, what other forward-looking scalability-related trait does MPI still lack; and how to reveal those traits and justify their importance? For instance, what happens when MPI runs out of memory to create its internal objects? What happens when an isolated peer becomes unresponsive in a very large group of processes? What facility does MPI provide for the custom resiliency needs of programs whose lifetimes are expected to include various kinds of partial failures? How do the non communication-related MPI features impact scalability?

1.3 Contributions

This dissertation addresses specific MPI issues which impact scalability. In Chapter 3, we show how efficient two-sided message progression in MPI falls short of the expectations in spite of the existence of autonomously progressing HPC network devices. The discussion is based on two-sided data transfers, which is the main communication model covered by the MPI literature on the message progression issue. The message progression issue shows up only in certain scenarios. On the one hand, there are the existing solutions which always miss at least one of the problematic cases. On the other hand, the solutions which cover all the problematic scenarios tend to adopt a brute force approach where their proposed fixes are applied even in the issue-free cases. As a consequence, the overhead associated with the fixes is incurred by the HPC application even when it is not justified. In Chapter 3, we approach the two-sided message progression with a proposal that brings the following contributions:

- A solution which can deterministically detect the specific scenarios where message progression fails to occur in a timely fashion. By applying a solution only in these scenarios, the overhead associated with our proposed fix is incurred only if it is justified.

- Memory-scalability thanks to the absence of a dedicated message progression thread for every single MPI process.

In Chapter 4 we show that message queue processing is very prevalent in MPI. Then we describe the impact of next generation HPC performance-oriented features on how often MPI message queues are solicited. We expose the scalability shortcomings of mainstream message queue designs and show how they render the resources available per CPU core inadequate for MPI processes at large scales. In particular, the chapter documents previous message queue designs which become quickly unusable for large jobs because they are scalable with respect to only one of speed of operation or memory consumption; but not both. Then, we present a novel multidimensional MPI message queue design which solves the aforementioned scalability problem by considerably mitigating the effects of job size on both speed of operation and memory consumption. To the best of our knowledge, the work of Chapter 4 represents the first two-fold scalable message queue architecture of MPI. The proposed message queue puts forth the following contributions:

- Considerable reduction of the length of linear searches in MPI message queues. We actually ensure that linear searches grow considerably slower compared to job sizes.

- Deterministic detection and skipping of very large portions of queue items which are guaranteed to be irrelevant during searches. This dimensionality-based narrowing down of the search leads to very slow search performance degradation as system and job sizes tend towards extreme scales. A special case of this contribution leads to the early detection of unfruitful searches. Therefore, major latency reductions can be achieved by preventing a receiving process from incurring potentially large search penalties to no avail.

- Amortized memory consumption and decreasing rate of memory consumption growth with respect to the queue size.

Chapter 5 presents an analysis of why MPI RMA falls short of meeting the one-sided communication expectations of the HPC community. We show the specific burdens that the MPI RMA synchronization model, which supports the epochs, puts on HPC applications. Chapter 5 brings the following contributions:

- Proposal of the first known MPI RMA improvement that completely removes the epoch serialization constraint; allowing one-sided communication lifetimes to effectively be nonblocking from start to finish.

- True decoupling between processes participating in MPI one-sided communications.

- First proposed correction to classes of latency issues inherent to MPI RMA. The serialized nature of MPI RMA epochs were the cause of six kinds of latency propagation issues first documented in [55] as the "MPI one-sided inefficiency patterns". Four out of the six inefficiency patterns had no remedy or workaround prior to the work covered in Chapter 5.

- Discovery of a new class of latency issue inherent to MPI RMA as specified in MPI-3.0, the latest version of the MPI standard. The new issue, which is also an inefficiency pattern is documented for the first time by the work covered by Chapter 5. A solution is brought to this new inefficiency pattern as well.

- Possibility to more efficiently realize in MPI a new class of HPC programs. In concrete terms, the proposed nonblocking RMA synchronizations allow HPC programs with dynamic and unstructured communication patterns to be expressed in MPI without large performance hit created by high contention levels.

Finally, we create in Chapter 6 unusual conditions of use and reveal many of the MPI limits at large scales. The experience leads to proposals to overcome these limits. The contributions of Chapter 6 are as follows:

- Presentation of the first known use of MPI as the network transport for extreme-scale distributed storage services.
- Demonstration of the scalability and safety tradeoff imposed by the blocking nature of non-communicating MPI routines. The proof leads to establishing the importance of nonblocking flavours of non-communicating MPI routines as the solution to certain cases of scalability issues that cannot be solved even if CPU cores were available in abundance in each node.
- Discussion of how the ability of cancelling communications can be a powerful primitive for custom resiliency mechanisms in software running on top of MPI. By making a practical experimentation-backed case for a cancellation-friendly MPI, we provide a conceptual approach to mitigating the MTBF issue anticipated for exascale systems.
- Creation by the storage service of conditions that naturally demonstrate at moderate scale the behaviour of major MPI implementations in situations of resource exhaustion. From the observed behaviours, we construct recommendations for MPI to sustain resource-demanding HPC jobs at large scales.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 presents background material on the typical software and hardware layers that support HPC jobs. An emphasis is put on MPI in its general structure as well as its specific features that are exploited later in the document. HPC network technologies are covered briefly as well. Chapter 3 gives an in-depth survey of Rendezvous-based message progression approaches in MPI and discusses methods that we investigated for improving message progress engines in certain scenarios. Chapter 4 provides a deep analysis of MPI message queues and shows why they are not adapted to the current and future job scales. Then, a new message queue design, with scalability as a primary concern, is described. Chapter 5 discusses MPI one-sided communications as it is currently specified. Then, the chapter presents the proposals brought by this dissertation to close the gap between MPI and the HPC community expectations. Chapter 6 presents our experience in using MPI in unconventional settings. The experience culminates in revealing a use case-backed list of scalability-oriented features that MPI currently lacks. Finally Chapter 7 concludes and introduces future work.

Chapter 2

Background

An HPC cluster is composed of a large number of compute nodes interconnected via a high speed network. From a hardware perspective (Figure 2.1), the memory hierarchy of a cluster has one additional level compared to a standalone workstation. For each node, the additional memory level is the remote memory of each of the other nodes; and since it is external, it is reached via the high speed network. Data transfer between memories of distinct nodes is expressed in the form of communication to the application-level messaging middleware (Figure 2.2), such as MPI, which is generic and network-device agnostic. The communication is fulfilled by the network device which is reached via a lower-level network-specific messaging middleware. In the stack, every level can resort to generic system software mechanisms such as memory allocation.

Each node can host one or several tasks; depending mostly on the number of CPU cores that it possesses. The unit of task execution in an MPI context is a process. From an Operating System (OS) point of view though, the unit of execution is a thread; an entity that requires exactly a single CPU core to execute without contention. By default, a process is single-threaded; that is, it possesses a single thread and requires a single CPU core. Processes can also be multithreaded in which case they host several threads and can exhibit parallelism internally. The following subsections describe the components of the stack as well as HPC network technologies.

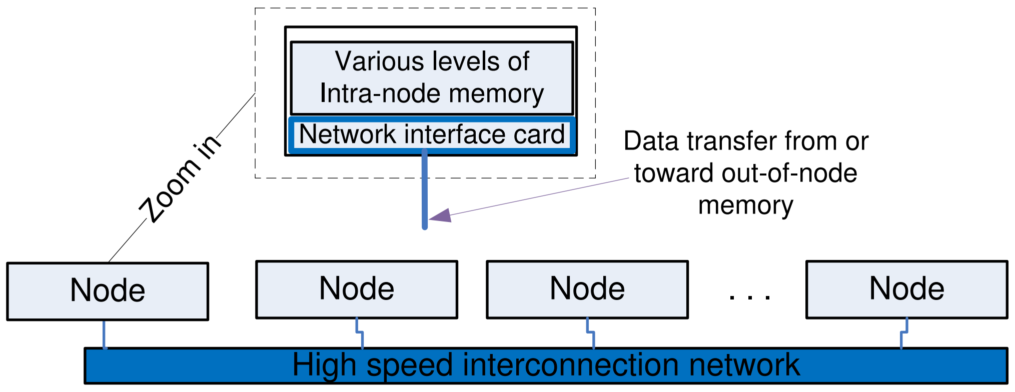

Various levels of Intra-node memory
Network interface card
Zoom in
Data transfer from or toward out-of-node memory
Node
Node
Node
. . .
Node
High speed interconnection network

Figure 2.1: HPC cluster: Intra-node and out-of-node memory view

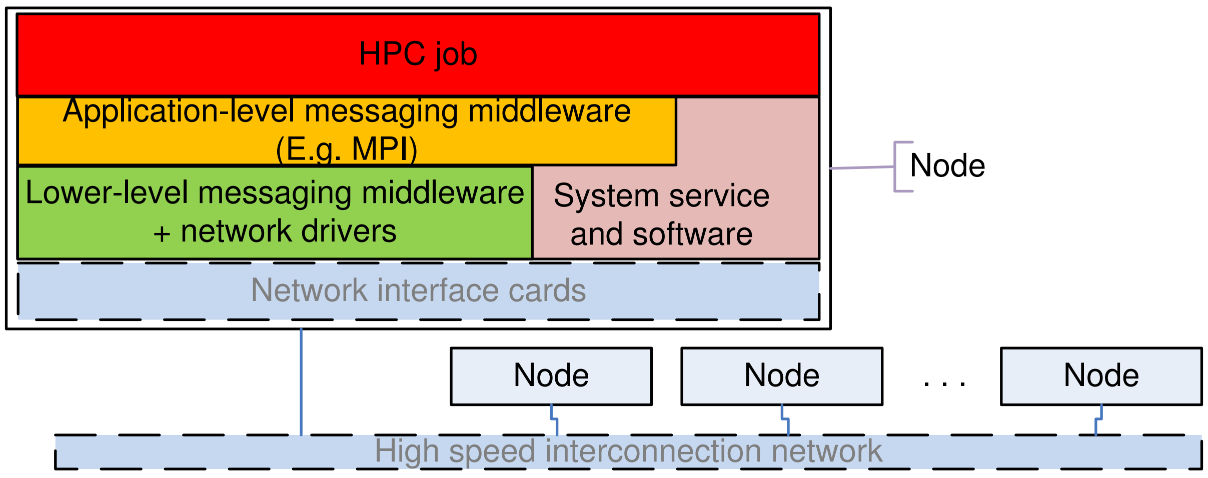

HPC job
Application-level messaging middleware (E.g. MPI)
Lower-level messaging middleware + network drivers
System service and software
Network interface cards
Node
Node
Node
. . .
Node
High speed interconnection network

Figure 2.2: HPC cluster: software stack view

2.1 Message Passing Interface (MPI)

Application-accessible messaging middleware are used to shield the HPC programmer from many system and network-level intricacies. In supercomputing, messaging middleware typically use the message passing model. MPI, which is the convergence of a few early messaging models, drives the advances in supercomputing thanks to its large prevalence as the HPC programming paradigm.

MPI is a standardized paradigm. The specification, maintained by the MPI forum [68], defines a language-independent Application Programming Interface (API) focused on portability. A binding is explicitly specified for C and Fortran; but implementations are possible in other languages. The current version 3.0 of the MPI specification has been released in September

2012 to supersede the previous MPI 2.2 standard. MPI-3.0 is the maturation of 18 years of effort following the release of the first MPI standard in 1994. MPI enforces a data-locality conscious programming by allowing only explicit communications between processes.

MPI manages processes in sets called *groups*. Groups are not usually manipulated in communications; instead, the programmer is presented with *communicators*, which define a communication scope or universe where each belonging process is uniquely identified by a *rank*. A rank is meaningful only in the scope of a communicator. In fact, the same process can have as many distinct ranks as the number of communicators it belongs to. Communicators can be of two types: *intracommunicators* and *intercommunicators*. An intracommunicator is conceptually no different from the group mapped to it. An intercommunicator is made of two groups that act exclusively as source or destination universes in a communication. The MPI application programmer can create communicators on-demand; and each user-created communicator can contain any non-empty subset of the existing processes. However, MPI provides the following predefined communicators which always exist by default in any job:

- `MPI_COMM_WORLD`: A communicator whose group encompasses all the processes.
- `MPI_COMM_SELF`: A communicator whose group is made only of the invoking process. Each process has its own `MPI_COMM_SELF`.
- `MPI_COMM_NULL`: The empty communicator.

An important mechanism of any MPI implementation is the *progress engine*, which ensures that any issued communication is handled to completion. The MPI progress engine takes care of transferring data as well as detecting and expressing transfer completion. Three MPI implementations have found widespread acceptance in the HPC community for being freely available at source-level for research and production. They are MPICH [69], MVAPICH [71] and Open MPI [79]. MPICH, maintained by the Argonne National Laboratory, is the reference implementation and the base source code for many other MPI implementations optimized for some specific high performance interconnects. Custom versions of MPICH are used on the large Blue Gene [28] and Cray supercomputers [36]. Many commercial MPI implementations such as Microsoft-MPI and Intel-MPI are also MPICH-derivatives. MVAPICH, maintained by the Ohio State University, is an MPICH-derivative which targets openly-specified modern network technologies such as iWARP [86] and InfiniBand [44]. Open MPI is an attempt to concentrate in a single distribution the advantages of three previous MPI implementations namely, LAM/MPI, LA-MPI and FT-MPI [26]. Although it has been around only since 2006, Open MPI has found an important level of adoption due to the widespread use of some of its building block implementations such as LAM/MPI [25].

In the rest of the dissertation, we stick with the case convention of the MPI specification. This convention uses all uppercase letters for language-agnostic MPI functions and constants. For instance the communicator creation routine is written `MPI_COMM_CREATE`. When a C binding is explicitly used, only the first five letters of routines are capitalized. For instance, the C version of the communicator creation routine is written `MPI_Comm_create`. Constants are still all uppercase in the C binding.

2.1.1 Datatypes and Derived Datatypes in MPI

Communications in MPI transfer arrays of data. The payload is thus always described by a datatype and a count. MPI provides many predefined datatypes which map to regular datatypes in programming languages such as C or Fortran. Examples of predefined MPI datatypes are `MPI_CHAR`, `MPI_WCHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_UNSIGNED_LONG`, etc. MPI even defines an address-sized type, `MPI_AINT`, that can be used in memory offset computations.

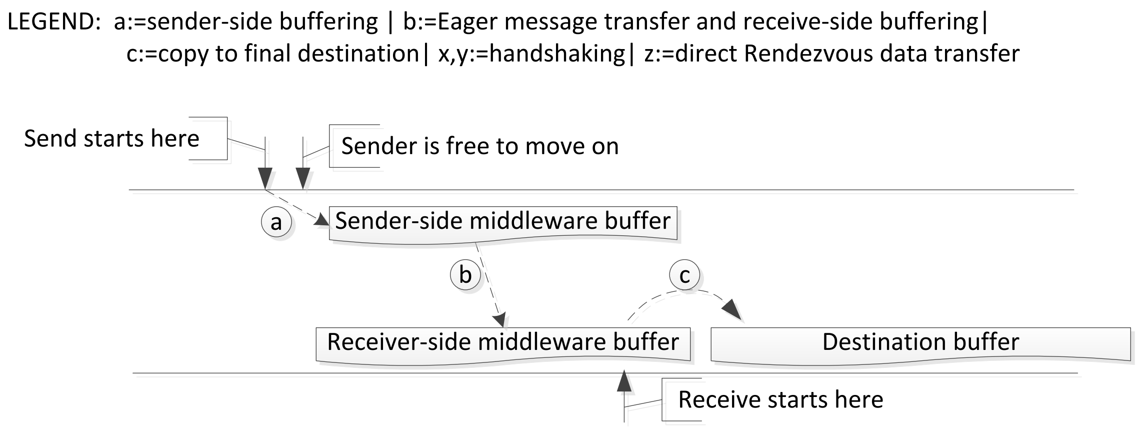
Sometimes however, certain messages cannot nicely be described as arrays of simple datatypes. Thus, MPI allows the programmer to define arbitrary datatypes by composing existing ones. Those custom types are referred to as *derived datatypes*. MPI allows the creation of contiguous and noncontiguous datatypes to describe messages with arbitrary gaps in their spanning memory addresses. A derived datatype can be made of other derived datatypes at an arbitrary level of nesting. Derived datatypes can be expressed as structures, vectors, *indexed* or *hindexed* types. An indexed datatype allows replication of an existing datatype, derived datatype included, into a sequence of blocks. The offset of each block in the new derived datatype is expressed as a multiple of the composing datatype size. An hindexed datatype is similar to an indexed datatype; except that the displacements are expressed in bytes.

2.1.2 Two-sided Communications

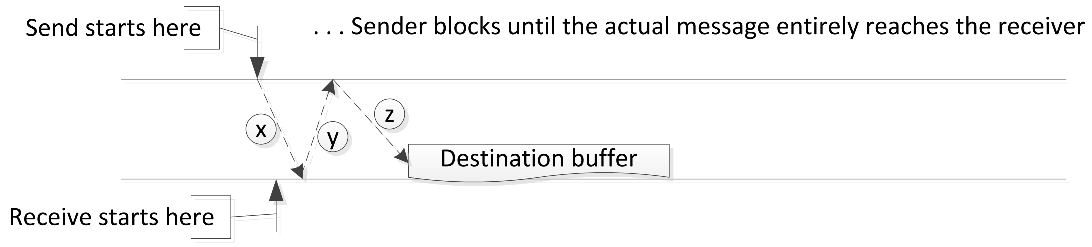
A two-sided communication, simply termed *point-to-point* or *two-sided*, involves only two processes and a send/receive semantics. A point-to-point message is uniquely identified at the sender side by the triple ⟨communicator, receiver_rank, tag⟩; and by ⟨communicator, sender_rank, tag⟩ at the receiver side. The triple provided by the receiver call must match the one provided by the sender for a two-sided communication to complete. However, instead of providing a specific source rank or tag, the receiver can provide `MPI_ANY_SOURCE` or `MPI_ANY_TAG` which are wildcards meant to respectively match any sender process in the same communicator or any tag. Wildcards allow the expression in MPI of communication patterns required by certain applications.

MPI specifies blocking and nonblocking send/receive operations. A blocking send (e.g., `MPI_SEND`) will not return until the data buffer being sent is safe to modify without compromising the content meant to be delivered. Safe to modify might mean that the data is buffered into some user-inaccessible system or middleware buffer and still pending transmission. Nonblocking sends/receives allow an MPI application to express communication/computation overlapping. A nonblocking send (e.g., `MPI_ISEND`) returns immediately but the programmer is responsible for guaranteeing that the application's sent data buffer is not modified until it is safe to do so. A blocking receive (`MPI_RECV`) blocks until the data is received. A nonblocking receive (`MPI_IRECV`) exits immediately but transfers to the programmer the responsibility of guaranteeing the integrity of the receive buffer until the reception completes. For both send and receive, the process can check, if required, when a pending nonblocking transfer has completed. Nonblocking two-sided must always be paired with one of the ***test*** or ***wait*** family of routines described in Section 2.1.5. Both blocking and nonblocking send operations are available in different flavours in order to either control system buffer use or to minimize the out-of-sync problems inherent to matching send/receive requests.

MPI implementations usually put forth two protocols for point-to-point message transfers. The first one, the *Eager* protocol, allows the sender to buffer the message and proceeds without waiting for a late receiver (Figure 2.3(a)). More generally, a message is said to be sent *eargerly* when the receiver is not notified of its arrival ahead of time. The Eager protocol is suitable only for small messages because it does an intermediate copy (step b in Figure 2.3(a)). Buffer copy is expensive for large messages. Additionally, middleware buffers are limited and cannot accommodate large messages. As a consequence, a handshaking (steps x and y in Figure 2.3(b)) is required for large messages so that the peer taking care of the transfer could know both the source and final destination addresses in order to perform a single and final message copy (step z in Figure 2.3(b)). The arrows x and y in Figure 2.3(b), called *control messages*, can bear various information such as the aforementioned buffer addresses. A control message is an internal message generated to ease MPI operation; it does not transfer application-level data. This handshaking-based approach of Figure 2.3(b) defines the second point-to-point protocol named *Rendezvous*.


LEGEND: a:=sender-side buffering | b:=Eager message transfer and receive-side buffering|
c:=copy to final destination| x,y:=handshaking| z:=direct Rendezvous data transfer
Send starts here
Sender is free to move on
a
Sender-side middleware buffer
b
c
Receiver-side middleware buffer
Destination buffer
Receive starts here

(a) Eager two-sided transfer


Send starts here
. . . Sender blocks until the actual message entirely reaches the receiver
x
y
z
Destination buffer
Receive starts here

(b) Rendezvous two-sided transfer

Figure 2.3: Two-sided message transfer protocols

Figure 2.3(a) purposely shows a late receiver so as to emphasize the usefulness of the Eager protocol. The receiver being early does not change the steps of the Eager protocol.

As for Figure 2.3(b), it depicts a very simplified version of the Rendezvous mechanism; the goal here is to simply distinguish between the two protocols. The actual engineering of the Rendezvous protocol can be complex; a deeper coverage of the Rendezvous-based point-to-point communication is deferred to Chapter 3. We emphasize that both Eager and Rendezvous data transfer approaches are middleware-level protocols; they are not visible at the application level.

2.1.3 Collective Communications

Collective operations, simply called *collectives*, involve all the processes in the communicator over which they occur. Collectives define three different semantics, namely, collective communication-computation (e.g., `MPI_REDUCE`), data distribution (e.g., `MPI_BCAST`) and synchronization (e.g., `MPI_BARRIER`). A collective communication/computation performs some distributed computation along with the data distribution. A collective synchronization does not transfer any application-level data; it simply guarantees that all the processes of a communicator have reached a certain point in the execution. Collective operations are also offered in different flavours characterized by whether the same amount of data is sent to all the participating processes. Some collective operations can be rooted, in which case a single process, called *root*, sends to or receives from all the other ones in the communicator. If a collective operation is not rooted, all the participating processes are senders and receivers at the same time. Collectives can also be blocking or nonblocking. Finally, a class of MPI communications, called *neighborhood collectives* allows collectives to occur over a group of processes whose layout or *topology* is known. The goal of neighbourhood collectives is to allow the MPI middleware to optimize the scheduling of the rounds of messages that make up the collective.

2.1.4 One-sided Communications

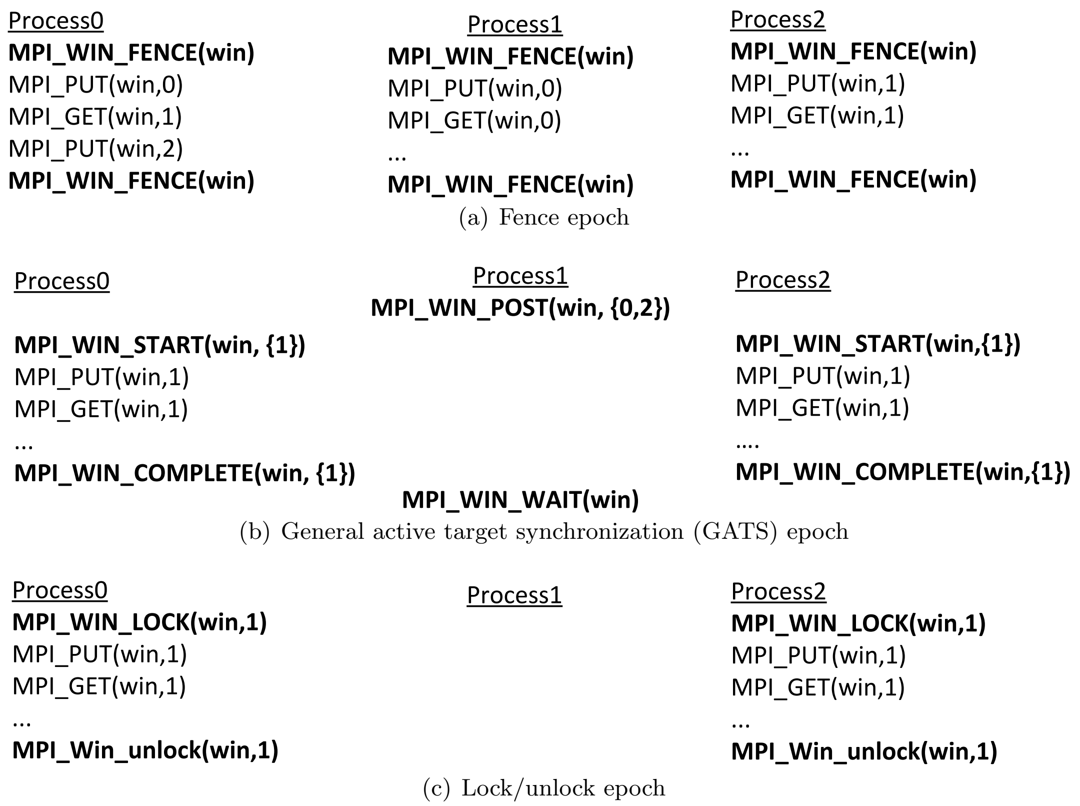
The MPI standard introduced its one-sided communication model in 1997. MPI RMA communications happen over objects called *windows* which define 1) the memory region that each process intends to expose for remote access and 2) the communication scope encompassing a set of processes.

In MPI RMA, the process reading or writing the remote memory is called the *origin* while the one whose memory is being acted on is called the *target*. MPI RMA occurs in critical section-like regions called *epochs*. An epoch is defined by a pair of opening and closing synchronization constructs. When an epoch is over, all its contained RMA communications are guaranteed to have completed for the concerned process. In one-sided communications, the origin process specifies all the communication parameters where the target process remains passive.

MPI RMA Synchronizations

MPI defines two classes of RMA synchronizations, namely *active target* where the target explicitly opens an epoch to match origin-side epochs; and *passive target* where the target does not even need to make epoch calls. There are two kinds of active target epochs; and they are *Fence* and *General Active Target Synchronization (GATS)*. All three kinds of epochs are shown in Figure 2.4. The `MPI_PUT` and `MPI_GET` calls shown in Figure 2.4(a) are examples of RMA communication calls; they are covered later in this section. In a fence epoch (Figure 2.4(a)), each participating process is simultaneously origin and target and can issue RMA communications to any peer in the communication universe covered by the window object. A single fence can also serve as both closing an epoch and opening the next one. For instance the sequence $Fence_0 - Epoch_0 - Fence_1 - Epoch_1 - Fence_2$ is correct; and means that $Fence_1$ closes $Epoch_0$ and opens $Epoch_1$ at the same time. For the sake of reference later in the document, we designate such a unique call which separates two adjacent fence epochs as a ***middle fence***. MPI allows *assertions* meant to guarantee that a fence call is not a middle fence. `MPI_MODE_NOPRECEDE`, when provided with a fence call, informs the middleware that the routine only opens a new epoch. `MPI_MODE_NOSUCCEED` tells the middleware that the current fence call only closes a previously opened epoch.

In a GATS synchronization, in a given epoch, a process is either origin or target but not both. In Figure 2.4(b), *Process0* and *Process2* are origins. Each of them opens an *access epoch* towards a group of targets. In Figure 2.4(b), the target group is only made of *Process1* which opens an *exposure epoch* for a group of origins. In a GATS synchronization, an origin can only communicate with the group of processes specified at epoch opening time (`MPI_WIN_START`).



```
Process0                  Process1                  Process2
MPI_WIN_FENCE(win)        MPI_WIN_FENCE(win)        MPI_WIN_FENCE(win)
MPI_PUT(win,0)            MPI_PUT(win,0)            MPI_PUT(win,1)
MPI_GET(win,1)            MPI_GET(win,0)            MPI_GET(win,1)
MPI_PUT(win,2)            ...                       ...
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_PUT(win,1)                                           MPI_PUT(win,1)
MPI_GET(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_PUT(win,1)                                      MPI_PUT(win,1)
MPI_GET(win,1)                                      MPI_GET(win,1)
...                                                 ...
MPI_Win_unlock(win,1)                               MPI_Win_unlock(win,1)
```

(c) Lock/unlock epoch

Figure 2.4: MPI RMA synchronizations (inspired by [105])

Similarly, a target can only be accessed by the processes specified in its epoch opening routine (MPI_WIN_POST). Those groups must be non-empty subsets of the overall group encompassed by the window object. The origin-side epoch is closed with MPI_WIN_COMPLETE while the target-side epoch is closed with MPI_WIN_WAIT. These epoch-closing routines are blocking and do not exit until all data transfers are completed locally for the concerned process. However, the target can test for completion with MPI_WIN_TEST which is nonblocking. A few assertions allowed for MPI_WIN_START and MPI_WIN_POST can enable the middleware to trigger some optimizations. For instance the MPI_MODE_NOPUT assertion, meant for MPI_WIN_POST, guarantees that the target epoch will only be read from; not written into.

Passive target epochs emulate shared memory. Since the target does not issue any synchronization call at all; it is truly unaware of being accessed for RMA (*Process1* in Figure 2.4(c)). An origin process in a passive target can lock a remote target either exclusively or in a shared fashion. An *exclusive lock* to a target is granted only if the target was not already locked via the same RMA window. Once granted, an exclusive lock does not allow any other process to lock the same target in the same window object. *Shared locks* allow a target to be locked by multiple peers simultaneously. A process can lock/unlock a single process with a pair of `MPI_WIN_LOCK`, `MPI_WIN_UNLOCK` calls. It is also possible to lock/unlock all the processes in an RMA window with a pair of `MPI_WIN_LOCK_ALL`, `MPI_WIN_UNLOCK_ALL` calls. Unlike `MPI_WIN_LOCK`, `MPI_WIN_LOCK_ALL` can only request shared locks.

Prior to MPI-3.0, RMA communications completion could not be detected before the closing synchronization call of the hosting epoch exits. In MPI-3.0, RMA communications occurring inside a passive target epoch can be completed without closing the epoch; by using one of a set of ***flush*** routines. MPI-3.0 also adds to passive target the support of request-based RMA communications whose completion can be detected inside the epoch via the nonblocking completion mechanism described in Section 2.1.5.

MPI RMA Communication Routines

There are three categories of MPI one-sided communications to allow an origin process to respectively store, load and accumulate data in the target. `MPI_PUT` and `MPI_GET` respectively fulfill the first two categories. An accumulate function uses the data provided by the origin as an operand; and the data at the destination in the target as a second operand. At the end of the accumulation, the operand in the target is overwritten with the result of a specified operation. The functions in the accumulate category are as follows:

- `MPI_ACCUMULATE`: This function performs an accumulation on operands which are vectors of arbitrary size. The function must specify the operation to perform in a set of predefined operations allowed by MPI. This function can be atomic when certain conditions are met.
- `MPI_GET_ACCUMULATE`: This function first fetches the unmodified target operand into an origin buffer which is disjoint from the origin operand buffer; and then performs an accumulate as described by `MPI_ACCUMULATE`.
- `MPI_FETCH_AND_OP`: This function behaves like `MPI_GET_ACCUMULATE`; but the operands must be a predefined MPI datatype; and the count must be 1. `MPI_GET_ACCUMULATE` can

operate on data of arbitrary size and is therefore heavyweight. In certain situations, the programmer might want to perform the equivalent the C ”+=”, ”-=”, ”++”, ”−−”, etc. on a single data of primitive types such as *int* or *double*. In those cases where the data is hosted remotely in the address space of the target, `MPI_FETCH_AND_OP` is provided as a faster lightweight alternative to `MPI_GET_ACCUMULATE`. This function is always atomic.

- `MPI_COMPARE_AND_SWAP`: This function swaps the target data with an operand provided by the origin only if the target value meets a certain condition. The function is atomic as well and can only operate on a subset of the predefined MPI datatypes, such as the integer and byte family of types. The count must be 1 as well.

For passive target epochs only, MPI provides request-based communication routines. Those routines are `MPI_RPUT`, `MPI_RGET`, `MPI_RACCUMULATE` and `MPI_RGET_ACCUMULATE`. MPI RMA communication functions are implemented as nonblocking. So, after `MPI_PUT` exits for instance, there is no guarantee that the data transfer has even started, let alone completed. As a consequence the origin buffer being operated on by `MPI_PUT` cannot be altered. The completion is guaranteed only after the epoch hosting the RMA communication is closed or after at least a subsequent ***flush*** function exits. Request-based RMA routines relax that constraint by returning a request which can allow selective completion detection by resorting to one of the ***test*** or ***wait*** family of routines (Section 2.1.5).

2.1.5 The Nonblocking Communication Handling in MPI

MPI nonblocking communications occur in two phases, namely *initiation* and *completion.* The initiation corresponds to the moment where the operation is issued. It is always nonblocking and returns a `REQUEST` object which is used later to detect completion. Completion can be detected in a blocking way by any of the ***wait*** family of MPI functions as follows:

- `MPI_WAIT`: Waits for a single request to complete.

- `MPI_WAITANY`: Waits for any single request to complete. This function takes an array of requests associated with a set of previously initiated operations. If more than a single request has completed, it arbitrarily reports one of them.

- `MPI_WAITSOME`: Waits for at least one request to complete. It reports all the completed requests. It takes an array of requests as well.
- `MPI_WAITALL`: Waits for the completion of all the requests passed in argument.

The nonblocking completion detection is provided as well through the ***test*** family of MPI functions. The ***test*** functions exit immediately and return a flag which reports the completion status. The ***test*** functions (`MPI_TEST{ANY|SOME|ALL}`) are provided in the same flavours as their ***wait*** equivalents. The ***test*** and ***wait*** family of functions are used uniformly for all kinds of request-based nonblocking functions of MPI.

2.1.6 Generalized Requests and Info Objects

The MPI specification provides *external interfaces* meant for extending the features natively offered by the standard. The main feature of the external interfaces is the *generalized requests* mechanism which allows the creation of arbitrary routines at application level by leveraging other MPI routines or function which are not natively specified by MPI. For instance, a programmer can craft a new collective adapted for some uncommon use case by piecing together other existing collectives or two-sided MPI functions. The custom routines can even be made nonblocking and leverage the usual ***wait*** and ***test*** family of MPI functions described in Section 2.1.5. The generalized request mechanism can be seen as a skeleton that can be instantiated and completed with user-specified hooks or function pointers to build an add-on. The add-on can then be used by the MPI progress engine to drive the custom functions created by the programmer.

MPI offers many other features meant for providing flexibility. One such features, the *info object*, is used to provide optimizing hints to MPI routines. Certain MPI routines have default behaviours which are justified most of the time. In a few cases, these default behaviours are too constraining and can therefore be overridden with a relaxing hint that is provided with an info object. An info object is of type `MPI_INFO`. It contains an unordered set of key-value pairs. The key can be used to define the name or type of a hint while the value is used to specify the actual hint or activation condition for the hint. Routines accepting info objects are predefined in the MPI specification. For each of those routines, the specification can sometimes provide predefined key-values pairs for the info object as well. The specification does not require an MPI implementation to support a predefined key-value pair. However, it does require implementations to either 1) silently ignore predefined key-values if they are not supported or 2) provide the exact behaviour specified for them if they are supported. Stated differently, a predefined key-value pair cannot be implemented with a behaviour different from the one specified in the MPI standard; and the MPI middleware cannot abort because an unsupported key-value pair is provided. A specific MPI implementation can also define new info object key-value pairs to provide optimization only relevant to the specific network device it runs on.

2.1.7 Thread Levels

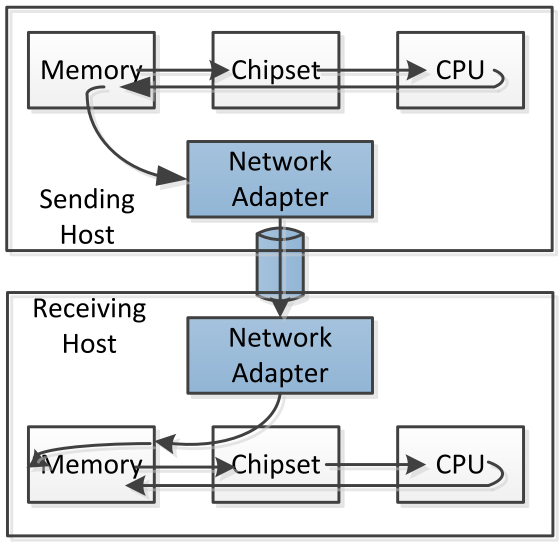
By default HPC jobs initialize the MPI runtime with the function `MPI_INIT`, which assumes that each process has a single thread. When processes in an HPC job require multithreading, a separate routine `MPI_INIT_THREAD` must be used instead to initialize the MPI runtime. The MPI specification defines four *thread levels* that the application can request from the MPI runtime at initialization time with `MPI_INIT_THREAD`. The thread levels inform the MPI runtime of the threading expectations of the application with respect to communications. The thread levels are monotonic and specified in increasing order as follows:

- `MPI_THREAD_SINGLE`: Each process has only one thread.
- `MPI_THREAD_FUNNELED`: Each process can be multithreaded but only the thread which initialized the MPI runtime can make MPI calls.
- `MPI_THREAD_SERIALIZED`: A process can be multithreaded but threads can only take turns to issue MPI calls.
- `MPI_THREAD_MULTIPLE`: Processes are free to multithread their executions as they see fit. MPI guarantees safety even in situations of concurrent MPI calls. However, the application programmer is responsible for preventing races when several threads make conflicting MPI calls.

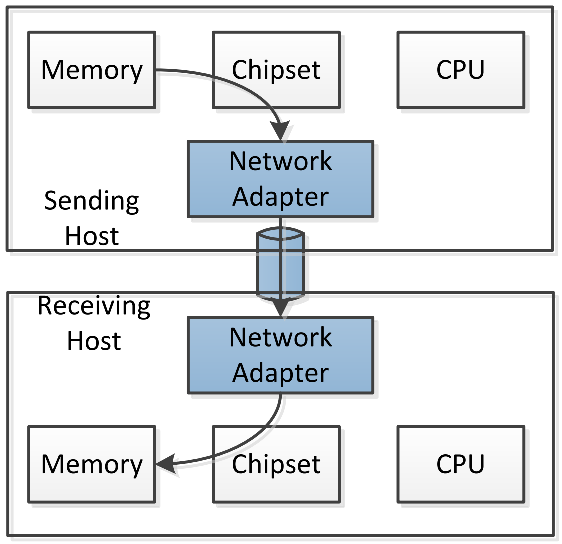
A high quality MPI implementation is expected to implement all the thread levels; but it is not a requirement. As a result, `MPI_INIT_THREAD` receives as input argument the thread level requested by the application; and outputs the actual thread level provided by the middleware. When an MPI implementation does not support the requested thread level, a call to `MPI_INIT_THREAD` provides the highest supported level right below the requested one.

2.2 High Performance Interconnects

High performance interconnects link the cluster nodes and allow them to communicate. They must offer low latency and high bandwidth. The latency of an interconnect is usually viewed as the fixed starting cost of transmitting anything from an end of the network to another one. It is common to judge the *0-byte* or *1-byte* latency of the network by measuring respectively how long it takes to ship an empty packet or a 1-byte data respectively. The bandwidth represents how much bits or bytes, on average, the network can transmit per second. There are two bandwidth metrics. The *signaling rate* is the raw bandwidth achieved on the physical link. The *data rate* is the bandwidth without the control headers meant for management tasks such as switching, routing or packet reassembling. The data rate is the maximum possible bandwidth that the application can experience. Latency is a key metric in HPC interconnects because most communications sent by HPC applications do not contain enough data to saturate the link for the bandwidth to matter. For instance, the performance of atomic operations like `MPI_FETCH_AND_OP` or `MPI_COMPARE_AND_SWAP` is very latency-sensitive. In order to provide low latency, HPC network technologies minimize as much as possible 1) the time it takes for the data to leave the application memory and reach the wire at the data origin side; and 2) the time it takes for the data to leave the wire and reach the application buffer at the receiving side. For that purpose, HPC networks provide mechanisms like *Remote Direct Memory Access (RDMA)* which accomplish *zero-copy* and *OS-bypass*. Zero-copy avoids intermediate copies of the data by giving the Network Interface Card (NIC) direct access to virtual memory (Figure 2.5(b)). In comparison, the widespread Transmission Control Protocol (TCP) first copies the data from the application buffer to some internal OS buffer; then into some device buffer before reaching the wire (Figure 2.5(a)). OS-bypass avoids the costly context-switch associated with using kernel-level drivers to reach the NIC. Furthermore, RDMA-enabled NICs usually do not require CPU involvement to stream data on the link. The node CPU, usually designated by *host* CPU, is therefore available for other computations; which could potentially be overlapped with ongoing communications.



(a) Conventional network device



(b) Autonomous network device

Figure 2.5: Zero-copy and CPU involvement (Adapted from [66])

Interconnects come in proprietary and commodity forms. A proprietary interconnect is built for a specific supercomputer or family of supercomputers. It usually does not have open specification. Examples of proprietary interconnects are TH Express-2 [21], the Gemini network [101], the Blue Gene/Q interconnect [15] and the Tofu network [67]; respectively for the four fastest supercomputers as of November 2013. Commodity interconnects support network devices that can be used in any machine which has regular workstation peripheral expansion buses like the Peripheral Component Interconnect Express (PCIe). Commodity NICs can require specific versions of those expansion bus (e.g. PCIe 3.x) for their full potential to be delivered. The common commodity interconnects in use in HPC are 10 Gigabit Ethernet (10 GigE), Myrinet [72], iWARP [86] and InfiniBand [44]. InfiniBand is the most prevalent commodity interconnect in the 500 most powerful supercomputers as of November 2013 [106].

For being the main interconnect used in the rest of the document, it deserves a bit of in-depth coverage.

2.2.1 InfiniBand: Characteristics, Queueing Model and Semantics

The InfiniBand Trade Association (IBTA) was created in 1999 to maintain the InfiniBand Architecture (IBA) specification [44]. The IBA specification covers everything from the physical interconnect to the routines exposed in software to interact with the IBA. The standard does not impose any API, but instead, it defines an abstract description of a set of functionalities that must be present to interact with the network interface. These abstractions called *verbs* can be represented with a combination of software and hardware means at the convenience of the manufacturer. IBA is characterized by an end-to-end latency that can be lower than 1μs. An InfiniBand network can operate at Single Data Rate (SDR), Double Data Rate (DDR), Quad Data Rate (QDR), Fourteen Data Rate (FDR) or Enhanced Data Rate (EDR) for respectively 2 Gb/s, 4 Gb/s, 8 Gb/s, 13.64 Gb/s and 25 Gb/s per lane. Additionally, the network can operate over 1, 2, 4 or 12 lanes for each data rate; pushing the maximum data rate to 300 Gb/s for 12-lane EDR.

InfiniBand allows the user to queue up *Work Queue Entries (WQEs)* that the network device executes when it becomes available. A WQE is either posted in a *send queue* or a *receive queue* depending on the transfer direction it describes. Send and receive queues are always created and managed as a single entity called *Queue Pair (QP)*. The network device, called *Host Channel Adapter (HCA)*, executes the WQEs in the order in which they are posted; but it does not wait for the completion of a WQE before consuming another one. Thus, the execution of multiple WQEs from the same or distinct queues can be simultaneously pending. When a WQE is completely progressed, the HCA might generate a *Completion Queue Entry (CQE)* that it posts in the adequate *Completion Queue (CQ)*. The client associates each send or receive queue with any CQ at will without restriction of sharing. CQs are typically polled to test for completion but the client can request an event generation upon the posting of select CQEs. Completion notifications can thus be consumed either on a polling or interrupt-based. Each receive WQE has a buffer associated with it to get the incoming data. With each QP having its own receive queue, a process must deal with n separate receive queues if it has n InfiniBand peers. It is not always possible to predict the number of incoming messages from each remote peer; and each process must either conservatively post receive WQEs in excess in each receive queue or risk exhaustion of the queue. To circumvent that issue, InfiniBand introduced the *Shared Receive Queue (SRQ)* mechanism that allows several QPs of a given process to pool receive buffers in the same receive queue; eliminating the need to guess the amount of transmission coming from each specific peer.

InfiniBand proposes two communication semantics. In the *channel semantics* embodied by send/receive operations, a party pushes data toward the other party which is thus responsible for determining where the data ultimately lands. The channel semantics requires the receiver to prepost receive descriptors (WQEs) in either the receive queue or the SRQ. The receive descriptor contains a buffer address, among other things. With n receive WQEs posted by the receiver, the sender can initiate a maximum of n channel semantics communications on the fly without any further receive-side buffer address information. The receiver cannot predict which receive descriptor will be used by each expected communication. As a consequence, it is not easy at application level to predict the address where the data of a specific communication will first be received in the destination virtual memory. In the *memory semantics*, the remote peer must first pin down the physical frames of a virtual address range meant to be exposed; and then communicate an access key to the initiating party. The memory semantics communication occurs one-sidedly. With a single key, the initiating party can issue any number of communications as long as the involved remote addresses are in the limits of the pinned memory frames associated with the key. This second operating mode, called RDMA, offers *RDMA Write* and *RDMA Read* respectively for storing to and loading from the remote memory. With the memory semantics, the non-initiating party knows at application level the range of virtual addresses where the data will first be copied to or from.

2.2.2 The Lower-level Messaging Middleware

MPI is layered on top of the lower-level messaging middleware. These lower-level middleware expose the features of the interconnect hardware as a software interface. Example of such middleware are the Parallel Active Messaging Interface (PAMI) [57] for Blue Gene/Q, the user Generic Network Interface (uGNI) [101] for Gemini networks, Myrinet Express (MX) [73] for Myrinet or the OpenFabrics Enterprise Distribution (OFED) [78] for iWARP and InfiniBand. Low-level messaging middleware can be implementations of openly specified APIs or behaviours that are not network technology-specific. For instance, although Portals [6] can be directly implemented in hardware, it can also have implementations on top of other lower-level messaging middleware such as OFED or MX.

2.3 Summary

This chapter covers HPC cluster architectures. We present the memory hierarchy as viewed by each process of an HPC job. We show how communication intervenes to allow access to the level of the memory hierarchy which is just below the physical memory of each node. The software stack which allows communication is also presented. We define MPI to the extent required by the upcoming chapters. HPC interconnection networks and their supporting messaging middleware are introduced as well.

We emphasize that clusters are specifically covered in this dissertation only for being the typical and dominant supercomputer architecture. The research presented in the following chapters is not tied to clusters; in fact all the upcoming proposals apply to any supercomputer capable of running MPI, that is, any contemporary supercomputer.

Chapter 3

Scalable MPI Two-Sided Message Progression over RDMA

Two-sided communications perform explicit matching, even at application level. A send in any process P_i requires a receive in some other process P_j, and vice versa. Message progression in situations of matched communications between two peers must involve both progress engines. With the Eager protocol of two-sided message transfer, the sender process does not wait for the receiver; the message is sent to some temporary internal buffer at the receiver side. As a result, the sender does not suffer any delay or latency propagation because of the receiver. For eagerly sent messages, this absence of receiver impact on the sender exists for both blocking and nonblocking communications. More rigorously, this absence of impact exists as long as the receiver-side middleware does not run out of internal buffer to temporarily host eagerly transferred data. In certain MPI libraries, internal buffer exhaustion could lead to job abortion. The MPI library could also set a flow control policy that forces the sender to wait for the receiver side to consume some of the backlog of eagerly stacked messages. In general, a sender waiting for a receiver with the Eager protocol can reasonably be considered a corner case. As for a receiver that resorts to the Eager protocol, no internal interaction occurs with the sender either, unless the aforementioned corner case occurs. Under normal conditions, the receiver just picks the message from inside the middleware if the message arrives before the receive-side process reaches the call site. If the receive call is made before the message arrives, the receiver must wait if the call is blocking. In summary, the Eager protocol does not require both progress engines to interact for the communication to complete at both ends.

The constraints are different for the Rendezvous protocol. In fact, because the Rendezvous protocol contains a form of handshaking, it requires at least one internal communication in each direction for the application-level communication to complete. As a result, the two progress engines are not simply involved in the communication, they must interact. The immediate consequence of this interaction is an increased potential for both processes to delay each other. In the blocking Rendezvous, both processes are trapped in their respective middleware until the communication completes. Each progress engine can react as soon as conditions are met for any piece of data transfer to occur. As a result, the different chunks of handshake messages and actual payload transfer occur in a timely fashion. With the nonblocking Rendezvous, the CPU is only briefly available to the middleware on the communication call site. Then if the handshaking cannot occur for the payload transfer to be triggered before the CPU returns to the application level, then the data transfer ends up being deferred unless a network device-based mechanism is available to remedy the need for control message arrival checking.

In Figure 3.1, we show an experiment where a communication is followed by a given amount of computation. The network device is assumed to be RDMA-enabled. As a reminder, RDMA allows the autonomous transfer of communication data by the network device; leaving the CPU available for the application. Without loss of generality, the communication is chosen to be a receive; that is, `MPI_RECV` or `MPI_IRECV` which is its nonblocking equivalent. The experiment is reproduced three times. The first time, `MPI_RECV` is used and as expected, the communication and the computation are serialized. In general, computation can occur only if the CPU is available at application level. In the first case of Figure 3.1, the CPU idles inside the middleware while the data transfer occurs via RDMA. In the second case, `MPI_IRECV` is used. However, because the data transfer did not start before the CPU went back to the application level, the communication ends up being serialized after the computation. The network device idles between the exit of `MPI_IRECV` and the entrance of `MPI_WAIT` even though there is a data transfer pending. Then, between the entrance and exit of `MPI_WAIT`, the CPU idles inside the middleware while the data transfer occurs. Finally, in the third case, the data transfer starts before `MPI_IRECV` exits. Then, while the CPU goes back to application level to perform computation, the data transfer continues and occurs in the same time frame, leading to communication/computation overlapping. As a consequence, the aggregate duration of the communication and the computation is lower.

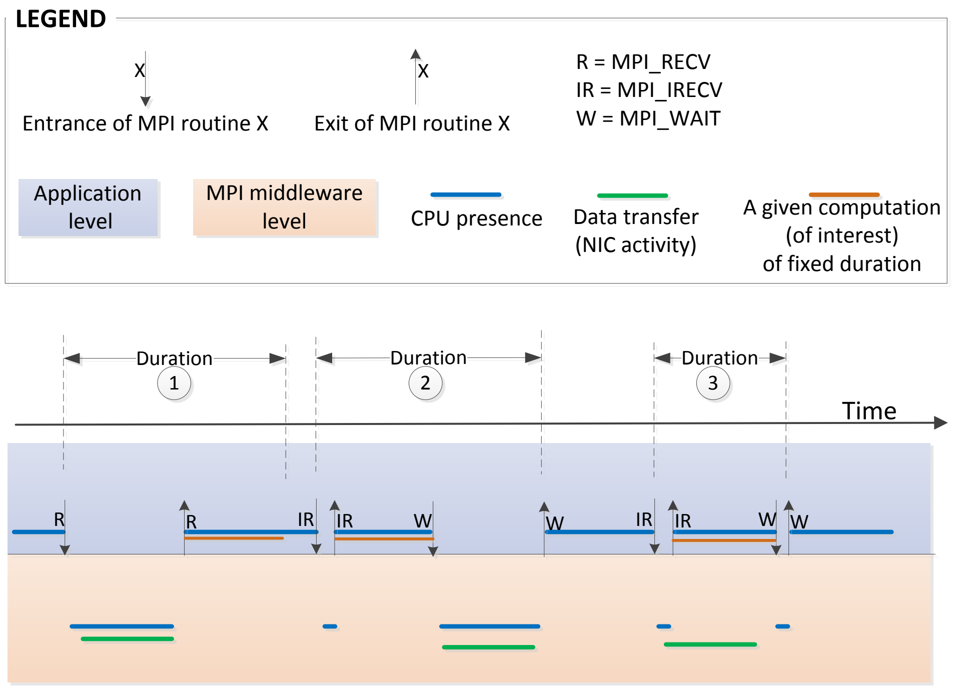


Figure 3.1: Serialized and overlapped communication/computation

The goal of any performance-conscious MPI distribution is to ensure as much as possible that the required phases of handshaking occur in a timely fashion; so as to realize the third communication/computation scenario shown in Figure 3.1. In reality, that goal is still usually not achieved. In this dissertation, we survey the various methods put forth to ensure communication/computation overlapping in the nonblocking two-sided Rendezvous protocol over RDMA-enabled network technologies. We discuss why some of these methods fail to cover all scenarios; and why some could easily become a scalability issue because of their resource consumption patterns. Then we introduce a comprehensive solution [117] that consumes resources only if required. The proposal is shown to be both effective and scalable.

3.1 Introduction to Rendezvous Protocols

3.1.1 RDMA Write-based Rendezvous Protocol

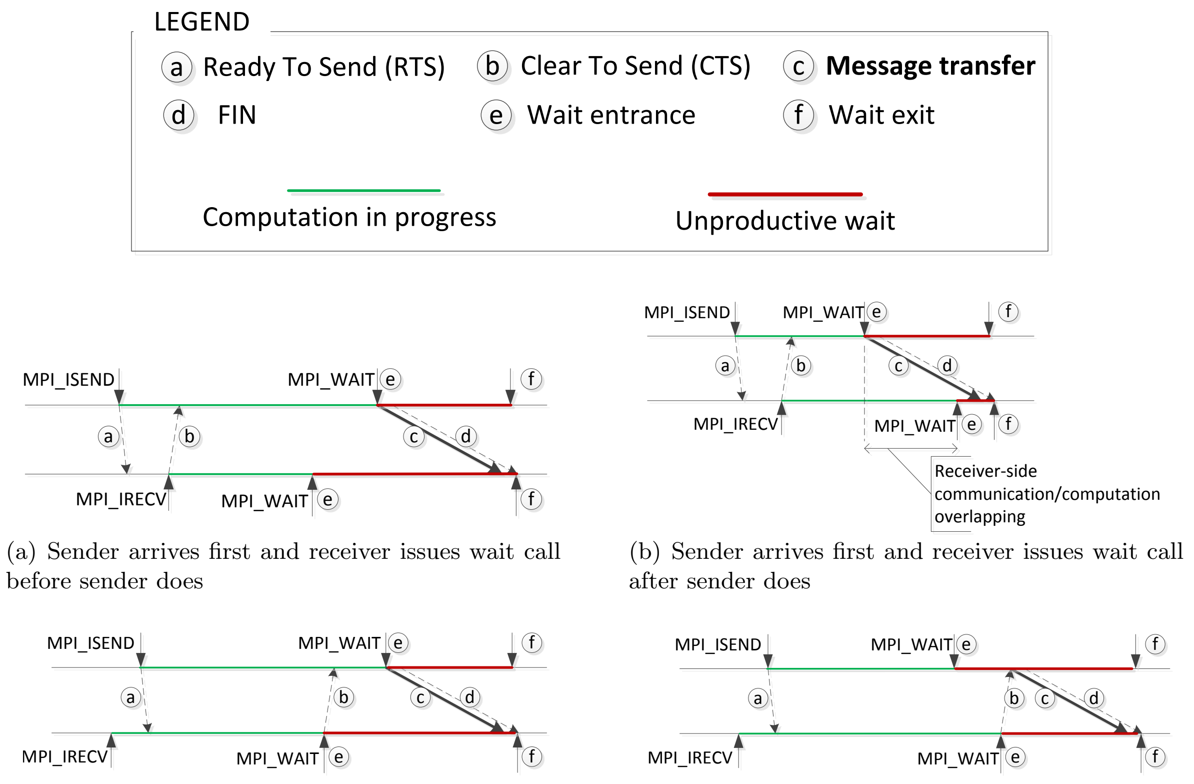
The most intuitive strategy is to have the sender RDMA-write the message into the receiver's destination buffer as shown in any of the diagrams of Figure 3.2. The process goes through steps *a* to *d* (Figure 3.2(a)) as follows:

- Step *a*: The sender side expresses its readiness by sending a Ready To Send (RTS) [85] control message to the receiver.
- Step *b*: Upon reception of RTS, the receiver side responds with a Clear To Send (CTS) [85] control message in which it specifies the address of the destination buffer.
- Step *c*: The sender side transfers the actual message to the receiver using RDMA Write.
- Step *d*: Finally, the sender side RDMA-writes a FIN control message to inform the receiver that the actual message is done transferring. The FIN message is an end-of-transfer notification.

Both InfiniBand [44] and iWARP [32] guarantee arrival ordering between several RDMA Write transfers posted back-to-back; and as per the RDMA specification of the Internet Engineering Task Force (IETF) [87], this ordering is to be expected from RDMA Write in general. As a result, the arrival of the FIN message necessarily means that the last byte of the actual message had reached the receiver.

Figure 3.2 shows the RDMA Write-based Rendezvous protocol with two different sender-receiver arrival scenarios and the wait patterns associated with each of them. Figure 3.2 shows that the sender can never accomplish any communication/computation overlapping because the actual message transfer always happens in its wait call. At the receiver-side, communication/computation overlapping can occur if and only if the sender comes first and issues its wait call before the receiver does. This scenario is shown in Figure 3.2(b) where the receiver-side overlapping is depicted to be the gap between the entrances of the two wait calls. When the receiver comes first, communication and computation are strictly serialized for both peers

(Figure 3.2(c), Figure 3.2(d)). We emphasize that the scenario described in Figure 3.2(b) can easily correspond to a complete receive-side communication/computation overlapping if the receiver's `MPI_WAIT` occurs substantially later than the sender's wait call. In such a case, the receiver would have no message transfer left to wait for; and its wait call would return immediately.



(a) Sender arrives first and receiver issues wait call before sender does

(b) Sender arrives first and receiver issues wait call after sender does

(c) Receiver arrives first and receiver issues wait call before sender does

(d) Receiver arrives first and receiver issues wait call after sender does

Figure 3.2: RDMA Write-based Rendezvous protocol

Th scenario depicted by Figure 3.2(b) does not make the RDMA Write-based Rendezvous any more compelling. In fact, the ability to proactively create communication/computation overlapping is correlated with the ability to exploit the window of opportunity between a nonblocking call and its associated wait call. The RDMA Write-based Rendezvous offers no such opportunity. In particular none of the two peers can create communication/computation overlapping for itself.

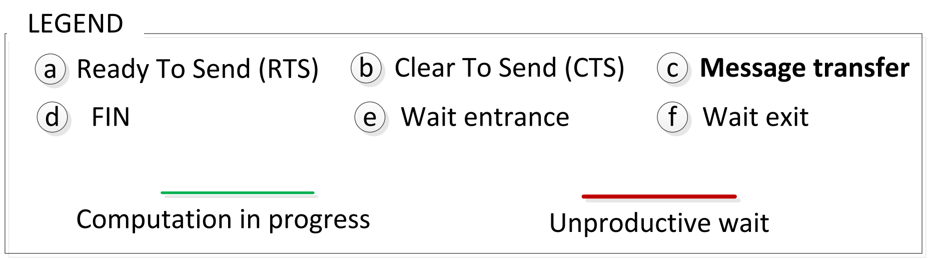
3.1.2 RDMA Read-based Rendezvous Protocol

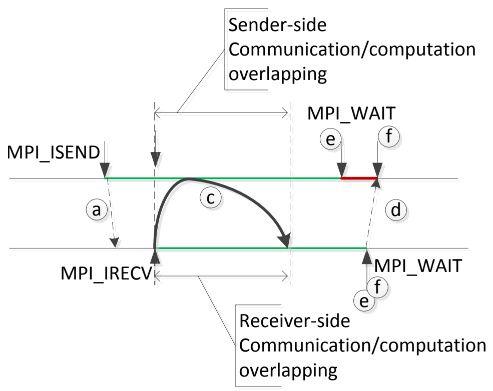
Figure 3.3 shows the RDMA Read-based Rendezvous. Instead of responding to the sender's RTS message with a CTS, the receiver directly pulls the actual payload using RDMA Read. The RTS control message conveys the address of the sender-side buffer. One of the advantages of the RDMA Read-based Rendezvous is the elimination of one control message; giving both peers the possibility to experience the actual message transfer before any of the wait calls is issued (Figure 3.3(a)). That ideal scenario occurs when the sender comes first; meaning that RTS awaits the receiver which can start the transfer before `MPI_IRECV` exits. If the receiver comes before the sender (Figure 3.3(b), Figure 3.3(c)), the receiver experiences strict serialization. The sender can still experience overlapping (Figure 3.3(b)) if its wait call occurs later than the receiver-side one (Figure 3.3(c)).

3.1.3 Synthesis of the RDMA-based Protocols and Existing Improvement Proposals

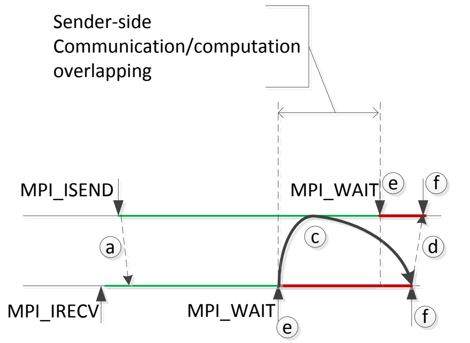
In all the previous Rendezvous schemes, the first control message is issued by the sender. A few proposals broke away from that traditional *sender-initiated* scheme by having the receiver sending the first control message of the handshaking. In a lightweight MPI implementation [80] meant for the Cell Broadband Engine [35], a receiver-initiated Rendezvous was proposed where the receiver makes the first move of the handshaking by sending a Ready To Receive (RTR) control message which contains the receiver-side buffer address. Then, in response, the sender directly RDMA-writes the message as shown in Figure 3.4. This approach can allow full overlapping when the receiver comes first (Figure 3.4(a)). Otherwise, no message progression happens until the sender enters its wait routine (Figure 3.4(b), Figure 3.4(c)); in which case there is no overlapping for the sender. In this case, the receiver can still experience overlapping if its wait call occurs after the sender-side one (Figure 3.4(c)). Notice that the receiver-initiated Rendezvous always uses RDMA Write because resorting to RDMA Read would always be suboptimal in this case.

In another proposal, both RDMA Read and RDMA Write-based protocols are multiplexed depending on the arrival order of the peers [84, 85]. We refer to these mixed protocols as *hybrid*.

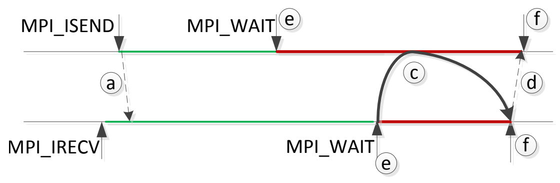



Sender-side Communication/computation overlapping
MPI_WAIT
MPI_ISEND
MPI_IRECV
MPI_WAIT
Receiver-side Communication/computation overlapping
a
c
d
e
f

(a) Sender arrives first



(b) Receiver arrives first and receiver issues wait call before the sender does

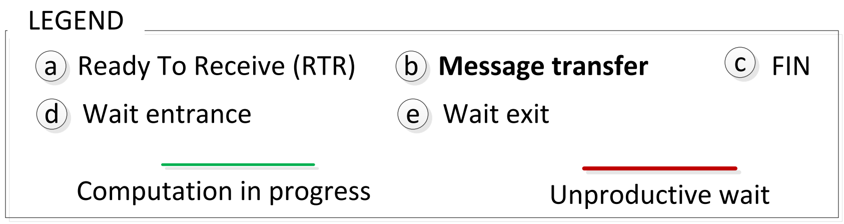


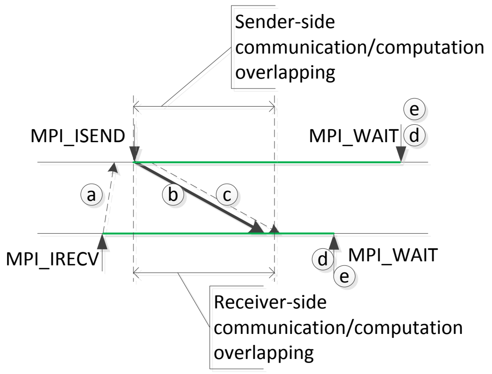
(c) Receiver arrives first and receiver issues wait call after the sender does

Figure 3.3: RDMA Read-based Rendezvous protocol

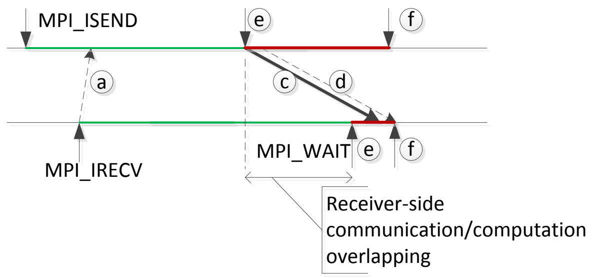
When the sender comes first, the RDMA Read protocol is used with a sender-initiated protocol (Figure 3.3(a)) and when the receiver comes first, the receiver-initiated scheme (Figure 3.4(a)) is used; guaranteeing an effective message progression in most situations. The Gravel library [16] also proposes the receiver-initiated Rendezvous among a set of other protocols that the user statically chooses from at job startup.

On top of mixing RDMA Read and RDMA Write as well as both kinds of initiators, a set of Rendezvous protocols [95, 96] was proposed where the sender can be wait-free when the receiver is too late. In particular, when the sender-side `MPI_WAIT` call is made and the receiver has not even issued `MPI_IRECV` yet, the sender locally buffers the message, notifies the receiver of a change of protocol and exits. Then, upon calling `MPI_IRECV`, the receiver RDMA-reads

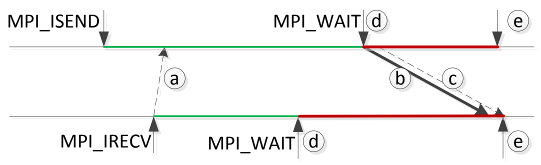



Sender-side communication/computation overlapping
MPI_ISEND
MPI_WAIT d
e
a
b
c
MPI_IRECV
d
MPI_WAIT
e
Receiver-side communication/computation overlapping

(a) Receiver arrives first



(b) Sender arrives first and sender issues wait call before receiver does



(c) Sender arrives first and sender issues wait call after the receiver does

Figure 3.4: Receiver-initiated Rendezvous protocol

the data from the sender-side internal temporary buffer. In another proposal [65], the buffering is done at the receive side by RDMA-writing the data in a pre-allocated receiver-side buffer when `MPI_IRECV` comes after the sender-side `MPI_WAIT`. In [65, 95], a message of any size can be buffered. While these bufferings allow the sender to shorten its overall communication time, they pose the issue of memory exhaustion at middleware level, especially for jobs made of a large number of processes. Considering the large size of the messages transferred with the Rendezvous protocol, these approaches raise a severe version of the concern expressed in [11] about memory exhaustion for Eager message buffering on systems meant to run large jobs. As a reminder, Eager messages are always small in size.

The receiver-initiated Rendezvous can only be speculative; meaning that the receiver can only assume that the communication will be based on the Rendezvous protocol. In fact, while the send call must specify the exact size of the data to be sent, the MPI specification only requires the receiver to specify a size big enough to get the message; that is, the receive size is allowed to be bigger than the actual message size. In certain applications, the receiver can only guess a lower bound of the message sizes that will come. As a result, a provided receive size can be beyond the Eager/Rendezvous threshold while the corresponding sender-side size is below that threshold. When that missed guess occurs, the receiver initiates a Rendezvous for a communication that ends up being eagerly performed.

A more serious issue with the receiver-initiated Rendezvous is the impossibility of coping with the MPI wildcards. As a reminder, the source wildcard `MPI_ANY_SOURCE` can be specified in a receive call to allow the receiver to expect a message from any peer in a communicator; and the tag wildcard `MPI_ANY_TAG` can be used to receive a message bearing any tag. For the receiver to initiate the handshaking, it must know which peer to send its RTR control message to. It must also know the tag that the sender intends to use. As a result, all protocol improvement proposals which resort to receiver initiation must fall back to a default, potentially inefficient protocol, when MPI wildcards are involved. Consequently, these protocols do not solve the lack of message progression in a comprehensive fashion.

Furthermore, which one of the sender or the receiver arrives first can safely be known only if the gap between the arrival times of the two peers is reasonably substantial. Actually, the receiver knows that the sender came first if it sees RTS waiting; otherwise, it assumes the opposite. However, RTS might be on its way when the receiver comes. In that case, the receiver assumes mistakenly that the other peer is late. The hybrid proposals refer to this scenario as *similar arrival times* for both peers. As mentioned in [95], none of the existing Rendezvous protocols is ideal when both peers arrive at similar times. In order to cope with this undesired case, the hybrid Rendezvous proposals can become entirely over-synchronizing [85, 95]. In particular, each control message has to be acknowledged to avoid race hazards in [85]. Having an oracle to predict the arrival order of each peer could help. A similar approach is considered in [65, 95] where a profile-driven mechanism is used to select the best protocols based on the relative timing of the calls of send/receive/wait routines. The proposal in [95] is based on the assumption that the MPI application at hand can be traced beforehand; so as to provide the various send/receive/wait relative call times. In [65] however, the profiling is done online inside the very application execution that is expected to use the profiling data. For that purpose, arrival times are gathered in each process and piggybacked to the remote processes in subsequent MPI calls. The profiling is done over two iterations and the predicted profile is decided. The profile-based proposals can suffer from non-deterministic events in the system running the application unless system noise [17, 59] is a tamed parameter; an unrealistic assumption. System noise is the set of perturbation caused by various activities that occur in the OS. These activities, created by OS facilities and services or potentially other applications, share the CPU, create cache disturbance or fill the translation lookaside buffer, among other things. As a result, they introduce unexpected and random delays in any program running on the same host. System noise makes it unrealistic to predict when a given instruction will be executed. Furthermore, the MPI program can and will usually have a different timing behaviour depending on the presence or absence of profiling instructions in the execution path. As a result, the profiling data might mispredicts the process arrival times that prevail when profiling is turned off. This last issue resembles the well-known change of behaviour when the same application is run in the same environment with and without debugging information in its binary.

3.1.4 Hardware-offloaded Two-sided Message Progression

All the previously mentioned work belong to the protocol improvement category. Hardware approaches have also been put forth to solve the Rendezvous issue. The TupleQ mechanism [53] associates with each InfiniBand SRQ a tuple $\langle$Communicator, rank, tag$\rangle$. This tuple is the same used by MPI to match messages between senders and receivers. If a sender is sending a message using a certain tuple for the first time, it requests the corresponding SRQ address from the receiver; the SRQ is created if it is not already available. The SRQ addresses are cached at the sender-side for future use. Then, each subsequent message is sent directly into the right SRQ without any control message. Of course, since RDMA is used, the scheme uses the HCA to progress the message. As a result, when a receive buffer is posted before a message sending is initiated, TupleQ achieves full overlapping at both sides. The key achievement here is the complete absence of control messages. The HCA itself blocks when the receiver buffer is not yet pre-posted and as such a software handshake becomes unnecessary. The first issue with TupleQ is the aforementioned blocking behaviour for late receivers. The second issue is that TupleQ cannot handle point-to-point wildcards. In those cases, a mechanism temporarily shuts TupleQ down and switches to the traditional inefficient point-to-point policies. When too many wildcards are used in an MPI program, this switching becomes expensive and TupleQ is definitely replaced by a non-hardware-based point-to-point protocol. TupleQ is not a comprehensive solution. Additionally, it presents a serious scalability issue because the number of SRQ required could explode. As the same tuple might come up in a subsequent communication, TupleQ does not free SRQs once they have been created. It helps to remind that InfiniBand queues are expensive resources whose abundant creation per process is documented as not scalable [52].

TupleQ is not the only InfiniBand-based message progression offloading. Some InfiniBand devices natively offer a feature, named CORE-Direct [31] meant to offload collective message progression from the CPU onto the NIC. As described in [45], CORE-Direct can be used for the Rendezvous protocol; but it requires a duplication of all the InfiniBand queue pairs in each process.

Message progression offloading exists beyond InfiniBand as well. With Myrinet [111] and the now discontinued Quadrics Elan [81], the offloading of message progression simply leverages a thread that runs right on the NIC.

3.1.5 Host-based Asynchronous Message Progression Methods

The asynchronous message progression resorts to an entity that progresses the message independently of the application thread that initiates the communication. The offloading approaches of Quadrics and Myrinet are actually examples of asynchronous message progression. When the NIC does not have a programmable thread, asynchronous means resort to vanilla host threads that run on the CPU.

In general, the frequent reference to threads for message progression as a silver bullet solution is mentioned in [39]. In particular, the experiments exposed in [39] depict the use of polling threads dedicated to spare CPU cores as a very promising solution to the message progression problem. However, the paper also recognizes that real life scenarios tend to discard the spare core avenue. In concrete terms, the spare core avenue implies that a node possessing n CPU cores hosts at most $n/2$ processes; so as to allow each process to have two threads, each hosted on its own CPU core. Those two threads per process are the default application thread and the dedicated message progression thread. However, common sense dictates to the user to always host n application threads on the n CPU cores of each node so as to maximize the parallelism of his job. As a result, the system ends up being oversubscribed with $2n$ threads competing for n CPU cores. Oversubscription has been shown to be detrimental, especially if the progress thread polls. As a consequence, while it is less performing than a polling thread enjoying a spare core, the use of interrupts becomes the other thread-based alternative when CPU cannot be dedicated. If the MPI processes are multithreaded instead, then the oversubscription is less severe than the $2n$ threads for n CPU cores case; but it is still present.

The interrupt-thread approach is based on RDMA Read, which is the transfer method for which it is optimal [102]. In fact, the interrupt is generated upon control message arrival and works best with the minimal possible number of those control messages. RDMA Read-based approaches are the only ones to allow a single control message before data transfer for sender-initiated Rendezvous. RDMA Write can allow a single control message as well if it is receiver-initiated; in which case it suffers all the issues related to the impossibility of handling MPI wildcards.

The usual criticism directed towards interrupts relates to their relatively high latency. With InfiniBand and OFED [78], there is also the impossibility of selectively generating the interrupts only in the cases where the message progression would not naturally happen. With RDMA Read, there are three possible receive scenarios, namely, 1) blocking receive; 2) nonblocking receive with the RTS control message coming before the receive call is issued; and 3) nonblocking receive with RTS coming after the receive call exits. Scenario 3 is the only case where the progression thread is required in order to prevent the message transfer from being deferred to the wait call. The interrupt happens at the receive-side but it is triggered from the send-side by setting a flag in the header of the RTS control message. Since the sender can predict neither the arrival time of the receiver nor the blocking nature of the receive call, it always sets the interrupt flag. As a result, the receive-side progression thread gets waken up for every receive operation. Triggering the progress thread equals incurring 1) an interrupt cost, 2) a context switch when the thread wakes up, and 3) either a lock to get into the progress engine, or another interrupt(signal) to the application thread to execute the progress engine [56]. That succession of events associated with each waking of the progression thread is too expensive to allow it to happen even in situations where it deterministically has no chance of producing any improvement.

3.1.6 Summary of Related Work

As stated in [39], and from what precedes, the message progression techniques can be grouped in a few categories: the protocol improvement approaches [16, 65, 80, 85, 95, 96], the hardware approaches [45, 53, 81, 111], and the host-based asynchronous progression methods [56, 102]. The protocol improvement approaches cannot deal with MPI wildcards and they can be over-synchronizing [85] because of the increase of control messages. They can also fail to be effective depending on the arrival pattern of the processes. The profile-guided versions of the protocol improvement approaches [65, 95] assume strict regularity in communication timings; an assumption that does not align with the behaviour of non-real time operating systems. In general, the main and persistent shortcoming of the protocol improvement approaches is that they do not offer comprehensive solutions. As for hardware approaches, they are either non-scalable [53] or restricted to a specific network device [45, 53, 81, 111].

Finally, asynchronous approaches that use host threads can be generic and applicable to any network device. They also offer comprehensive solutions. When they resort to polling, they lead to oversubscription [39]. When they resort to interrupts, they suffer from the impossibility of distinguishing between the problematic scenarios and those where overlapping naturally occurs without any help [102, 56]. This brute-force approach does not scale either.

3.2 Scenario-conscious Asynchronous Rendezvous

As a solution to the absence of message progression in the Rendezvous protocol, we investigate an asynchronous Rendezvous approach that is completely scenario-conscious. For being asynchronous, the approach covers all possible scenarios . Plus, it requires no specialized hardware and is not linked to any particular network device like the offloaded methods. As shown in Figure 3.2, Figure 3.3 and Figure 3.4, not all combinations of process arrival patterns suffer a lack of autonomous message progression. The solution proposed in this section selectively operates in the only cases where the progression does not naturally start.

In order to introduce the design, we establish in general that a sufficient condition for an all-time effective autonomous communication progress requires three mechanisms:

- A *carousel* meant to ferry messages from one end to another without any external propelling force.
- A *watchdog* to check the availability of control messages or transfer conditions at either end of the carousel.
- A *trigger* to kick off the carousel. This includes dropping/picking up messages on/from the carousel if applicable.

Streaming the data, as done by the carousel is the most expensive aspect of the communication. With RDMA being the carousel, only small amounts of CPU activities are required to fulfill the watchdog and the trigger. Consequently, we depart from the use of dedicated threads and put forth a solution that leverage existing ones. In fact, we try to steal those small amounts of CPU from a thread that is supposed to run at application-level inside an MPI process.

Our design [117] is built around RDMA Read. As previously mentioned, RDMA Read fulfills the Rendezvous protocol with the least amount of control messages before the start of the data transfer. In particular, only the receiver gets a control message. Additionally, it is the receiver that triggers the data transfer with RDMA Read. Consequently, only the receive side would require a watchdog-trigger tandem to operate. As for how to sneak into the application thread, source code transformation before or during the compilation phase is not a viable option because it would give no control over the watchdog activity frequency. In fact, no information is available on individual instruction execution durations. Unfortunately, breaking down each source-level instruction into smaller ones is not always possible; especially when they involve deeply nested library function calls. At the binary-level, a certain control can be expected over instruction execution times; but tracking data sharing and race issues at that level is extremely challenging. Thus, we elect not to tamper with the instructions. Instead, we start our investigation with a fiber-like [88] construct that would coexist with the regular instruction flow inside the thread. Fibers are distinct execution flows that could be hosted in the same thread. They take turn non-preemptively to use the same CPU as long as the hosting thread is running. Fibers can be viewed as the system-level implementation of coroutines, which are language-level concepts. Fibers are natively supported on Windows NT but not on Linux.

3.2.1 Design Objectives

At this point, we know that the watchdog will poll, the Rendezvous protocol would be RDMA Read-based and that a separate instruction flow would be required to realize the watchdog. As shown in Figure 3.3(a), the RDMA Read-based Rendezvous protocol is already known to allow independent progress and overlapping when the RTS message reaches the receiver before `MPI_IRECV` is issued [85, 95, 102]. Thus, the design needs to care only about the case where the receive call is issued before RTS arrives.

We designate by Application Execution Flow (AEF) the flow made of the default instructions of the thread; and by Parasite Execution Flow (PEF) the flow made of the instructions we are sneaking into the thread at the middleware-level to realize the watchdog and the trigger. Before `MPI_IREV` is issued, the thread only has AEF. Then, if and only if `MPI_IRECV` does not find RTS in its unexpected message queue, PEF is spawned to periodically execute the watchdog until RTS is found and the RDMA Read transfer is triggered. Activating PEF only in this case is a form of scenario-consciousness. This whole mechanism is described in Figure 3.5. One can notice that the proposed solution is used only when it is required by a nonblocking receive because, unlike the OFED and InfiniBand interrupt approach, the triggering is decided at the receive side. Furthermore the Rendezvous is sender-initiated. As a result, the receiver does not have to know or guess the source rank or the tag of the communication. There is therefore no issue with wildcards.

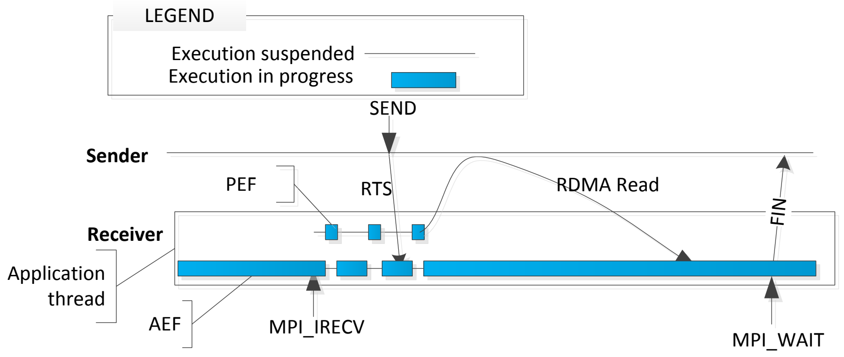


Figure 3.5: Parasite execution flow-based message progression at receiver side for two-sided communications

PEF and AEF behave like fibers and take turn exclusively to consume CPU cycles. For a single pending `MPI_IRECV`, PEF is active for at most the duration of the possible overlapping period. The overlapping period ends when any of the ***wait*** family of routines is called for the receive request. It also ends when the transfer completes; even if the wait call is not issued yet. As shown in Figure 3.5, PEF is guaranteed to die at most at its next turn after RTS arrives. There is only a single PEF in the application process no matter the number of pending nonblocking receives. If it was not active, PEF is spawned by the next `MPI_IRECV` that misses its RTS. Then, any subsequent `MPI_IRECV` that executes before PEF dies just adds a progress request to the queue of currently watched receive requests. PEF dies when the last expected RTS is found or when any of the ***wait*** family of routines is called for the last pending request. In order to allow only nonblocking receives to trigger PEF, we propagate a flag at the entrance of `MPI_IRECV` calls. By so doing, we realize another scenario-consciousness. In comparison, the interrupt-based approaches that are decided at sender-side operate even for blocking receives.

3.2.2 Asynchronous Execution Flows inside a Single Thread

A thread can be considered created with a single default fiber just as a process is created with a single default thread. That default fiber (AEF) runs the instructions that are programmed for the thread at application level. The default fiber does not share its thread with any other instruction flow until a new fiber (PEF) is explicitly created in due time by a call to `MPI_IRECV`. In an ordinary multi-fiber environment, AEF and PEF would take turn to consume CPU cycles by willingly yielding the processor from time to time. Fibers are thus inherently cooperative and synchronous. The switching from a fiber to another one is done by issuing an instruction whose position is deterministically known in the execution flow. Yielding the CPU when PEF is running is trivial because the specific and exact watchdog instructions that it runs are known by the MPI middleware. The same is not true for AEF because its instructions are provided and decided by the MPI application programmer; and those instructions are arbitrary. There is no easy means of inserting CPU yielding instructions in AEF without raising the same issues associated with the code transformation avenues mentioned earlier. From what precedes, there are two issues with considering AEF as a fiber. First, it will never yield by itself and second, it will not yield at the desired frequency or moment. An asynchronous means is thus required to disrupt the flow of AEF. The use of timer interrupts appears to be an immediate candidate. The timer would periodically preempt AEF to transfer the CPU to PEF. Then, when done, PEF willingly re-transfers the CPU to AEF. By resorting to this semi-preemptive mechanism, we depart from the pure fiber concept; and at the same time we lose some of the beauties of fibers. In particular, disrupting AEF takes away the peace of mind associated with the complete absence of race condition that fibers offer.

The design is realized on x86_64 Linux. Hosting PEF and AEF in the same thread is a concept that does not map well on the features offered by Linux. In particular, sneaking PEF into the thread seems feasible only by altering the OS kernel. With the regular features offered by Linux, we single out signal delivery as a potential means for performing the AEF to PEF transition.

A signal is an event-like interprocess communication common on the broad Unix family of operating systems. Signals are in theory supported by all Portable Operating System Interface

(POSIX)-compliant operating systems. A process can send a signal to another process or to itself. A signal can also be initiated by the OS or by various events such as keyboard combinations (e.g. Control+C). There are many kinds of signals and each has an associated default action. A process can either set a mask to defer the delivery of a specific signal, define a custom handler in order to take a user-defined action upon the delivery of the signal or simply set a flag to ignore the signal even if it is delivered. This freedom of management is possible with most signals types. When a signal is sent to a process, the OS delivers it asynchronously and interrupts the execution flow of one of its threads. In certain cases, a signal can be delivered to a specific thread in a target process.

For the AEF to PEF transition, we resort to SIGALRM which is delivered periodically after a custom timer expires. For performance reasons, the disruption must be lock-free and avoid all contention between AEF and PEF. PEF executes inside a defined SIGALRM handler. There is no programmer-accessible data access issue because PEF is disabled when AEF is about to call the progress engine which is the only critical section of interest. The disabling is done by setting an `ignore` status for SIGALRM. There is therefore no risk of AEF being preempted when inside the progress engine. As far as Linux signal handlers are concerned, hidden corruptible data are composed of I/O functions as well as data manipulated by dynamic memory allocations. These data are not touched in PEF.

We provide the following environment variables to control the AEF/PEF alternations. A period *ppef_period* to specify the time between two consecutive turn taking of PEF; a phase *ppef_phase* to specify the delay before PEF takes turn the first time after a new request is queued; a frequency decay *ppef_freq_decay* to specify a multiplicative factor of the period after each turn taking; a turn limit *ppef_max_turns_per_req* to specify the maximum number of turn taking after which PEF gives up progressing a request. All those parameters are reinitialized and reapplied every time a new request is posted for an already existing PEF. The tuning space defined by the aforementioned parameters is very large. Without a prior knowledge of the application at hand, tuning is thus a difficult task, and the subject of future studies. However, we propose a two-step rule of thumb that serves as the default set of parameters. The first step is optimistic. It assumes that the application is well-behaved; meaning that the peers are balanced enough to exhibit only small gaps in communication call times; and a small value of the phase (e.g., $2\mu s$ or $5\mu s$) should help offsetting that gap. If RTS does not arrive in the time frame of the phase, the rule enters its second step where it becomes more and more pessimistic after each PEF turn that does not hit. The pessimistic step is realized by a period that is multiplied by *ppef_freq_decay* after each turn. The pessimistic step gives up a bit of reactiveness each time in order to limit the overhead imposed on the receiver by very late RTS control messages. On our test system, the rule uses a phase of $2\mu s$, a period of $10\mu s$ and a frequency decay of 2.

3.3 Experimental Evaluation

Our experimental setup is a four-node InfiniBand cluster. Each node is equipped with two quad-core 2GHz AMD Opteron 2350 processors and 8GB of memory. The nodes run Linux kernel 2.6.32. The network devices are Mellanox ConnectX QDR cards and switches. The MPI distribution is MVAPICH-1.2rc1, which is a version with thread-based asynchronous Rendezvous progression. The Eager/Rendezvous protocol threshold is 9180 bytes. The tests compare the MVAPICH-ASYNC method, which implements the interrupt-thread approach, with our proposed PEF method. RGET designates the default RDMA Read-based Rendezvous in MVAPICH. RGET does not resort to any asynchronous agent.

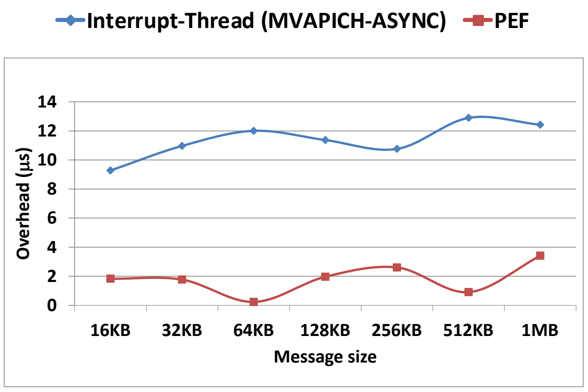
We emphasize that microbenchmarks are used to test and observe in isolation a very specific behaviour. As a result, it is common practice to report microbenchmark results over large numbers of iterations; so as to abstract away the differences that might occur from execution to execution. In comparison, applications cannot be executed in controlled environments; and it is neither necessary, nor realistic to seek to remove differences in behaviours in an application across multiple executions. It is a common practice in the field to report application results over small numbers of execution iterations.

3.3.1 Microbenchmark Results

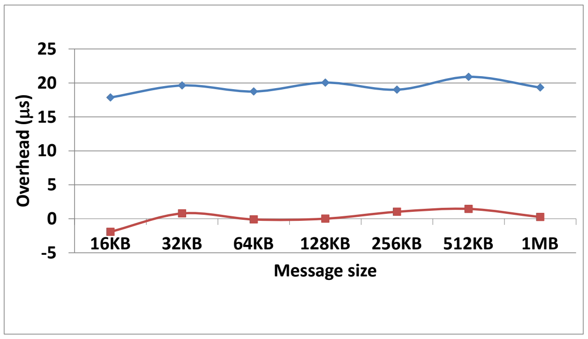
Overhead Tests

Figure 3.6 shows the latency overhead of MVAPICH-ASYNC and PEF compared to RGET. Barriers are used to force arrival orders for nonblocking tests. No computation is inserted between the nonblocking calls and their associated wait calls; thus RGET does not suffer any message progression issue. As a result, RGET always yields the smallest possible latency and can therefore be used to compute the latency overhead of the two other Rendezvous protocols. Each latency data is measured by averaging 1000 pingpong tests. A pigpong test measures the duration of a round trip communication between two peers. All overheads are computed relative to RGET. The scenarios "blocking receive" (Figure 3.6(a)) and "nonblocking receive, sender arrives first" (Figure 3.6(b)) show that the overhead of PEF oscillates around $0\mu s$ while MVAPICH-ASYNC exhibits on average $10\mu s$ and $20\mu s$ per message transfer. These two cases do not present any message progression deficiency. The scenario-consciousness of PEF justifies its absence of overhead in these scenarios where MVAPICH-ASYNC is triggered even though there is no need and no room for improvement. The measurements have some small noise component that justifies why Figure 3.6(b) shows a slightly negative overhead for 16KB.

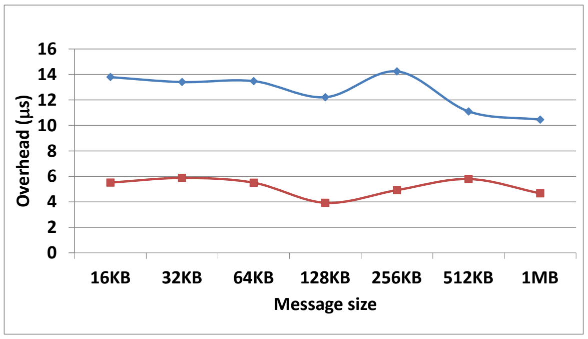
When the receiver is nonblocking and early compared to RTS (Figure 3.6(c)), a progression help is required to avoid deferring the message transfer to the wait call. Figure 3.6(c) shows that PEF exhibits an overhead of approximately $5\mu s$ in this scenario. MVAPICH-ASYNC exhibits on average $12\mu s$ in this scenario. While the different overheads shown by the PEF curves in the three cases can be explained by its scenario-consciousness, the reason of the difference in overheads might be less obvious for the MVAPICH-ASYNC method. We remind that the interrupt-thread mechanism, as implemented by MVAPICH-ASYNC, goes through three steps when it triggers. First, the interrupt delivery wakes up the progression thread. Then in the second step, the progression thread sends a signal to the main application thread that handles the message progression; and then goes back to waiting for another interrupt in step three. However, the signal sent by the MVAPICH-ASYNC thread is temporarily ignored application-wide every time the application thread is inside the MPI middleware as in the



(a) Blocking receive



(b) Nonblocking receive; sender arrives first



(c) Nonblocking receive; receiver arrives first

Figure 3.6: Receiver-side latency overhead of interrupt-based threading (MVAPICH-ASYNC) vs. PEF. (RGET is the reference)

case of blocking receives. Ignoring the signal is important to avoid race conditions. The aforementioned second step is thus less expensive in Figure 3.6(a) than in Figure 3.6(b) and in Figure 3.6(c); and so is the communication latency of MVAPICH-ASYNC. We emphasize that the signal is still sent in Figure 3.6(a), but it triggers no handler because it is ignored.

The latency of RGET is approximately the same in Figure 3.6(a) and Figure 3.6(b). Due to the less expensive latency of MVAPICH-ASYNC in Figure 3.6(a), its overhead is smaller than in Figure 3.6(b) where the signal is delivered. As for Figure 3.6(c), its associated sequence of calls is `MPI_IRECV`, `MPI_BARRIER`, and `MPI_WAIT`. RGET, in Figure 3.6(c), yields a higher latency compared to Figure 3.6(b) because its message transfer starts in `MPI_WAIT` instead of `MPI_IRECV`. This higher RGET latency mitigates the overhead of MVAPICH-ASYNC in this third scenario, making it less than in Figure 3.6(b). The low $5\mu s$ overhead of PEF in Figure 3.6(c) can also be attributed to that mitigation.

Communication/computation Overlapping

We designate by ε the duration of a nonblocking two-sided communication whose application-level payload has size 0. ε can be seen as the fixed cumulative latency of issuing `MPI_ISEND` and `MPI_WAIT`; or `MPI_IRECV` and `MPI_WAIT`. We also consider a single activity made of a communication of duration c_m and a computation of duration c_p. The overall duration d of the activity when there is no communication/computation overlapping is expressed as:

$$d = \varepsilon + c_m + c_p \tag{3.1}$$

If communication/computation overlapping is achieved, the overall duration is instead expressed as:

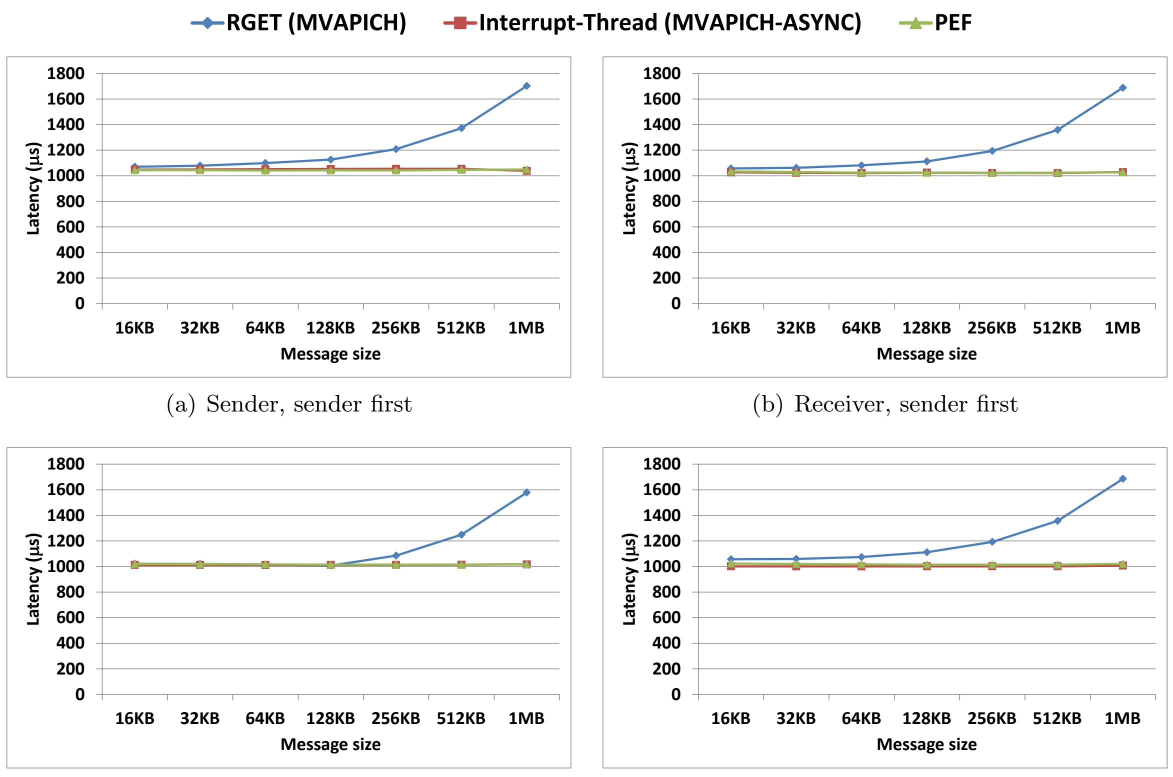
$$d = \varepsilon + max(c_m, c_p) \tag{3.2}$$

As per Equation 3.2, if a constant c_p is chosen to be always larger than c_m for various message sizes, then d is constant and becomes:

$$d = \varepsilon + c_p \tag{3.3}$$

The communication/computation overlapping test results are shown in Figure 3.7. $c_p = 1000\mu s$ so as to be larger than the largest c_m which is less than $750\mu s$. The computation is done simultaneously on both sender and receiver sides to keep the CPU busy at application-level in both peers. Communication/computation overlapping is assessed by observing if the curve of d is decoupled from the message size (Equation 3.3) or if it shows a positive slope when the message size increases (Equation 3.1). The two scenarios of sender coming first (Figure 3.7(a) and Figure 3.7(b)) and receiver coming first (Figure 3.7(c) and Figure 3.7(d)) are represented.

One can notice for both the sender and the receiver, and for both arrival orders, that the interrupt-thread and PEF approaches allow communication/computation overlapping; and PEF is on par with the interrupt-thread approach in spite of being better in terms of overhead generation (Figure 3.6). In comparison, the default RGET protocol does not allow communication/computation overlapping. In fact, it is already known that RGET can proactively create communication/computation overlapping only when the sender comes first (Figure 3.3(a)); so the observations in Figure 3.7(c) and Figure 3.7(d) are according to the expectations but the observations in Figure 3.7(a) and Figure 3.7(b) are not. The lack of communication/computation overlapping in Figure 3.7(a) and Figure 3.7(b) is due to a choice made in the MVAPICH implementation. An arrived control message could wait in the message queue at middleware level. A control message could also be waiting at InfiniBand verb level, in which case it has to be retrieved by the progress engine. The InfiniBand verb-level retrieval is not done for the RTS control message by MVAPICH with the RGET implementation.



(a) Sender, sender first

(b) Receiver, sender first

(c) Sender, receiver first

(d) Receiver, receiver first

Figure 3.7: Communication/computation overlapping of RGET (MVAPICH), interrupt-based threading (MVAPICH-ASYNC) and PEF

3.3.2 Application Results

The results in this section are generated with the NASA Advanced Supercomputing (NAS) application suite [74]. A NAS application is executed in a given class that defines the size of the dataset that it manipulates. Classes are named with alphabet letters. In the application graphs, we use the convention $X.Y.n$ to designate the NAS application X, run in class Y in an MPI job of n processes.

We stated that PEF is better than the interrupt-thread mechanism (MVAPICH-ASYNC) by being scenario-conscious. In particular, blocking receives and nonblocking receives for which RTS arrives before the receiver are scenarios where the interrupt-thread mechanism deals an undue penalty to the application. This means that unlike PEF, it is risky to use the interrupt-thread mechanism on HPC applications whose communication profiles are not known. Such a restriction is very limiting especially because HPC consumers do not usually care about middleware or communication profiles.

For our tests purpose, it is not easy to force an arrival order-based scenario in applications. Nevertheless, we can present the impact of both PEF and the interrupt-thread mechanism on blocking receives. The results, using the NAS LU application, are shown in Figure 3.8. NAS LU is a Lower-Upper Gauss-Seidel solver that uses only blocking receives. Figure 3.8 shows that the interrupt-thread (MVAPICH-ASYNC) mechanism degrades the communication by more than 8% for 4 processes and by more than 22% for 32 processes. In comparison, PEF shows a degradation of less than 0.2% and 0.9% for 4 and 32 processes respectively. Except from a mere branch to check the blocking nature of the receive, our proposal does not execute any instruction in this scenario where PEF is not triggered at all. As a result, those tiny percentages are more tributary to noise than actual penalty. From the trend observed in Figure 3.8, we can conjecture that the interrupt-thread mechanism would deal larger overheads to larger jobs; exacerbating the issue at larger scales.

The other application tests for nonblocking receives are performed for:

- NAS BT: Block Tri-diagonal solver
- NAS CG: Conjugate Gradient

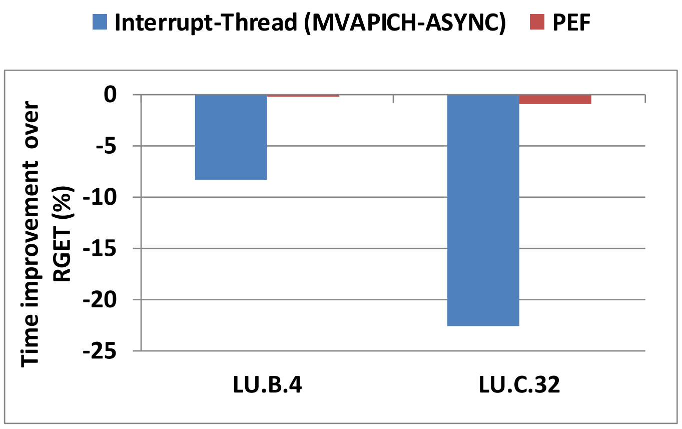


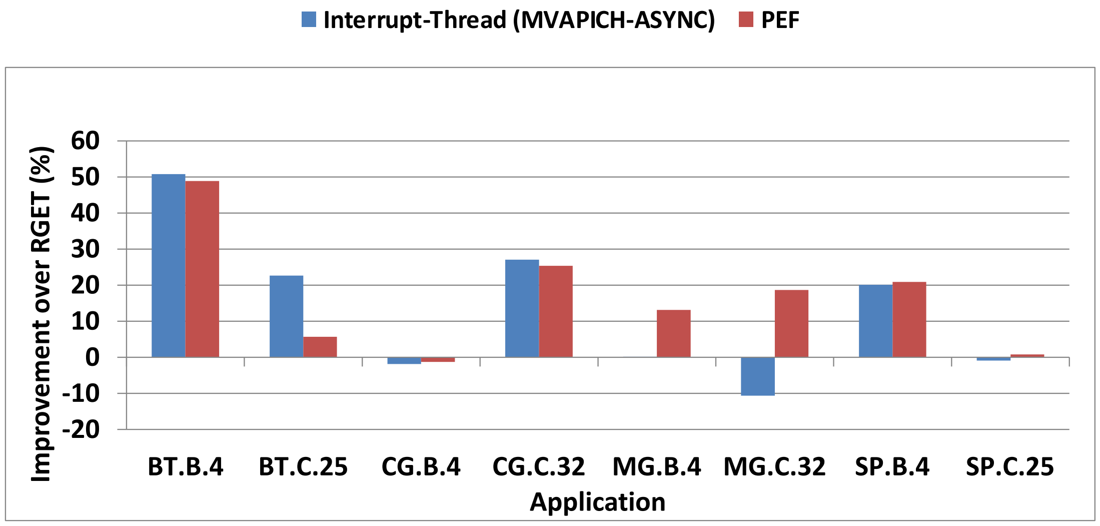
Figure 3.8: Impact at receive side on the communication time of a job that resorts to blocking receives

- NAS MG: Multi-Grid on a sequence of meshes

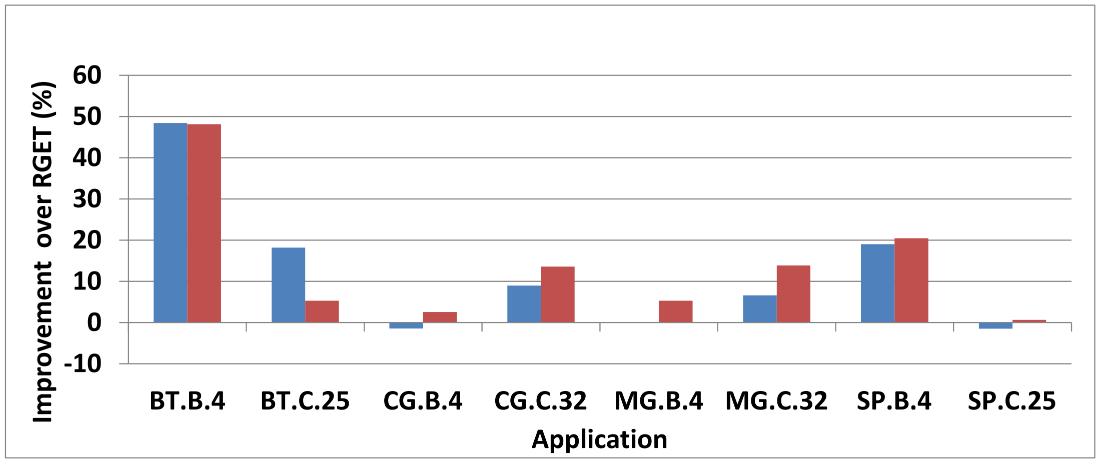
- NAS SP: Scalar Penta-diagonal solver

BT and SP can only be executed over square numbers of processes; and can reach a maximum of 25 processes on our 32-core cluster. All the tests are averaged 6 times. Figure 3.9(a) and Figure 3.9(b) respectively show the overall wait and communication times of the executions. We first observe that both mechanisms yield similar performances for some of the applications; namely, BT.B.4, CG.B.4, CG.C.32, SP.B.4 and SP.C.25. We also observe that for BT.C.25, MVAPICH-ASYNC shows better results while PEF performs better for MG.B.4 and MG.C.32. Once again, the key observation remains the absence of a general trend of PEF paying its scenario-consciousness by underperforming compared to the interrupt-thread mechanism.

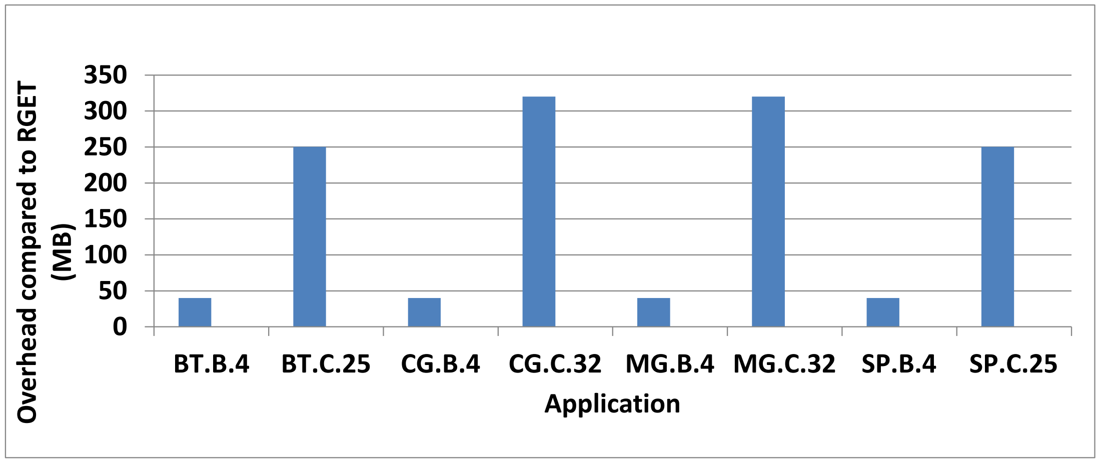
In general, it is important to remind that the actual nonblocking communication progression is handled by RDMA. The two means being compared here just help in triggering the transfer in a timely fashion. As a consequence, when the RTS lateness is reasonable, nonblocking communication progression performance while RDMA is available tends to be a two-state variable swinging between a maximum possible performance and zero. This explains well the behaviour observed for BT.B.4, CG and SP in Figure 3.9(a) and Figure 3.9(b).



(a) Wait time improvement



(b) Communication time improvement



(c) Memory footprint overhead

Figure 3.9: Nonblocking receive-based application benchmark results

However, application behaviours can sometimes create some differences in the performances as shown by BT.C.25, MG.B.4 and MG.C.32. If RTS arrives mostly early, MVAPICH-ASYNC would tend to perform poorly compared to PEF. PEF on the other hand would tend to be outperformed by MVAPICH-ASYNC when RTS mostly exhibits slightly large delays. In these cases, PEF tend to be less reactive than MVAPICH-ASYNC. One can notice that very large delays are a performance killer, no matter the Rendezvous mechanism used, as they end up voiding the reactiveness advantage as well. In particular, very large delays, when they exist, tend to be the dominant communication latency component; making the actual transfer time insignificant.

Finally Figure 3.9(c) shows the memory footprint overhead of each mechanism. The memory footprint overhead is how much each mechanism adds to the resident set size of each application when compared to MVAPICH-RGET. For all the applications, PEF shows an insignificant overhead, reaching 128KB in the worst case for 32 processes. In comparison, the interrupt-thread mechanism shows more than 320MB in the worst case for 32 processes. Each x86_64 Linux thread adds 10MB to the resident set size. Once again PEF seems to be the better mechanism when scalability is a concern. In Figure 3.9(c), the memory overhead of PEF is not visible because it is too small compared to the memory overhead of the interrupt-thread approach.

With respect to the memory overhead of the interrupt-thread approach, potential ways of diminishing to some extent the amount of memory that a Unix/Linux thread adds to the working set of a process could come in mind. One approach is to alter the stack size of all the threads created by the user with a Linux resource limit management facilities such as the `ulimit` command. This approach makes a restrictive assumption about the resource needs of all the threads created by the user on all the nodes. Another approach is to alter the code of the MPI middleware to control the creation parameters of the interrupt thread. This second approach also is not always possible because it assumes the availability of the sources of the MPI middleware as well as some knowledge about its internal functioning. In practice, there are only limited and unpractical solutions to the memory footprint issue associated with the interrupt-thread approach.

3.4 Summary

The purpose of nonblocking communications is to mitigate the latency of data transfer. However, for that mitigation to occur, the data transfer must not be deferred to the blocking wait call that completes the nonblocking communication. In two-sided communications, the deferral can occur on Rendezvous-based data transfers if certain control messages of the handshaking are not discovered before the nonblocking call exits. A category of previous work proposed various alterations of the Rendezvous protocol to solve the issue; but they always miss at least one scenario where data transfer deferral can occur. Hardware solutions exist that solve the issue in a comprehensive manner, but they are tied to specific network devices. As a network-agnostic comprehensive solution, a host thread can be leveraged to check for control message availability. That approach, named asynchronous message progression, can either resort to a polling thread or to a thread that is awaken upon interrupt generation. Since MPI users do not leave spare cores to dedicate to middleware-level activities, the polling approach leads to oversubscription. The interrupt approach leverages non-dedicated CPUs; but it is oblivious of the cases when it is not required to trigger; making its associated overhead difficult to justify in many scenarios. The asynchronous proposal investigated in this dissertation seeks to avoid these unjustifiable overheads by being scenario-conscious.

We realize our proposal by forcing the application thread itself to handle Rendezvous-backed point-to-point RDMA operations even when it is already plunged in a computation. We do so by having a Parasite Execution Flow (PEF) coexisting with the Application Execution Flow in the same thread. While the interrupt-thread mechanism is triggered from the sender side; our mechanism is entirely controlled by the receiver itself. For having a precise knowledge of the need or not for progression help, the PEF mechanism can thus be triggered only when it is required. As a consequence and unlike its interrupt-thread counterpart, every overhead that the PEF mechanism imposes to the application is associated with a situation that requires improvement. Plus, PEF is substantially lighter than the interrupt-thread mechanism for memory footprint overhead. On top of its positive overhead control and memory footprint behaviours, the PEF proposal is on average as efficient as the interrupt-thread mechanism in terms of performance improvement for the application studied in the dissertation.

Each Rendezvous protocol discussed in this chapter sends at least two control messages, including the FIN notification. In the nonblocking versions of the protocol, the first set of control messages, RTS, CTS and RTR can all reach their destination without the concerned process being actively waiting for them inside the MPI middleware. Like the middleware-level control messages, application-level messages can also reach a peer that is not actively waiting for them. What happens to those messages that are not being waited for? The answer is simple; they are queued! With many communications expressed at application level, the middleware could have to deal with proportionately many payload messages and proportionately many control messages; and when scales grow those queued messages become less trivial to process. Retrieving a specific message in a queue that services a hundred processes is very easy for a CPU. The task could still be trivial with a thousand processes; but with jobs of hundreds of thousands or even millions of processes, the scalability issues quickly become obvious. If retrieving a message queue item is part of internally fulfilling a communication, then the communication can noticeably slow down. Knowing that resources per process are limited, how does an MPI process hosts and manage large message queues anyway? These questions are investigated in the next chapter where we show that message queues are a central piece of MPI middleware; a piece that grows even further in importance when scales grow.

Chapter 4

Scalable Message Queues in MPI

Sequoia, one of the most powerful petascale systems [106] has 1,572,864 CPU cores hosted in 98,304 compute nodes. These resource levels are expected to grow even larger in the exascale computing era. Little issues which are benign or unnoticeable on small systems can become unforgiving at large scales. An example of one such issues resides in the processing of message queues in MPI.

Message queues are required in MPI libraries to cope with the unavoidable out-of-sync communications [4, 49]. A minimum of two message queues are required, both at the receive-side, to allow MPI communication operations. They are the *Unexpected Message Queue (UMQ)* and the *Posted Receive Queue (PRQ)*. When a new message arrives, the PRQ must be traversed to locate the corresponding receive queue item, if any. If no matching is found, a *Message Queue Item (MQI)* is queued in the UMQ. Similarly, when a receive call is made, the UMQ must be traversed to check if the requested message has not already (unexpectedly) arrived. If no matching is found, a new MQI is posted in the PRQ. These message queues are required because senders are not required to synchronize with receivers before sending small (Eager type) or control messages.

In MPI, the minimal search key in any message queue is the tuple $\langle$contextId, rank, tag$\rangle$. In certain implementations such as Open MPI [79] it is a proper superset of that tuple. ContextId designates the communicator inside MPI. The rank is the source process rank for a PRQ item, and the receive process rank for a UMQ item. As for the tag, it is useful whenever the same process (sender or receiver) can have more than a single pending message or MQI in any queue. Internal predefined tags are also used by MPI libraries even for communication models, such as collectives and RMA, which do not require a tag argument in the API.

It has been observed that the message queue length can grow in proportion to the job size [12, 13, 14, 49]. Message queues are solicited in point-to-point, collectives and even modern RDMA-based implementations of RMA operations [89], which marginally resort to point-to-point operations at middleware-level. Actually, the UMQ is used so frequently that it has been qualified as the most crucial data structures in MPI [49]. Message queue operations must therefore be fast at large scales as they are on the critical path of most MPI communications. This performance requirement is a condition for MPI to remain a viable and scalable HPC communication middleware. Scalability, however, is two-fold. With the growing processor core density per node, and the expected lower memory density per core at larger scales, a queue mechanism that is blind on memory consumption behaviour could be as harmful as one that quickly becomes unacceptably slow when job sizes grow. If a message queue architecture can remain reasonably fast at 1 million CPU cores while its memory consumption becomes prohibitive at 0.1 million CPU cores, then its scalability is effectively limited to 0.1 million CPU cores.

In this dissertation, we propose a multidimensional MPI message queue management mechanism which exploits rank decomposition to considerably mitigate the effects of job size on both speed of operation and memory consumption [118, 119]. We compare the behaviour of the proposed design with two other reference message queue approaches which perform extremely well with respect to either memory or speed scalability but could become quickly unusable for large jobs for not being two-fold scalable. We show in particular that the proposed design is able to not only offer unbounded message queue processing speedup, but in certain situations, even outperform the reference memory-scalable message queue approach in terms of memory footprint.

4.1 Related Work

Various flavours of message queue implementations exist outside the MPI middleware. MPI over Portals implementations are mentioned for which the UMQ and the PRQ exist in NIC memory [14]. The Cray MPI [30] implementation offloads most of its receive-side matching operations on the Portals network infrastructure [6], which it is layered on. Though, according to the Portals specification draft [6], a resource exhaustion risk exists for large message queues. MPI libraries over previous versions of Portals abort the application when queue resource exhaustion occurs. In the current version 4 of Portals, this situation is handled by dropping packets; and the end result is a slowdown.

Quadrics [81], which is a discontinued commodity HPC network device, and Myrinet [107] also support MPI message matching by offloading it onto their NICs. They resort to on-NIC threads and memory. For the InfiniBand [44] interconnect, TupleQ [53] has been proposed where a verb-level SRQ is created for each $\langle$contextId, rank, tag$\rangle$ tuple to match messages in hardware. As previously mentioned in Section 3.1.4, TupleQ was not meant for queue processing but for handling the point-to-point Rendezvous protocol in hardware. For a receiver process P, the SRQ requirements of TupleQ grow in $O\left(c \times r \times t\right)$, with c , r, and t respectively being the number of contextIds, ranks and tags which have ever reached P. The issue with TuppleQ resides in both the cubic growth pattern and the resource consumed by SRQs which are kept forever even when they are never reused again. TupleQ is therefore not suitable for large applications, even when there is no queue buildup.

A hardware proposal has also been reported in [110], where a NIC has been modified to process the queue operations with an associative list processing unit. The NIC is equipped with a local static random access memory. The tests presented in their work are only simulations made on a custom stripped-down implementation of MPI-1.2. Finally, a NIC-associated accelerator has been proposed [103], to store message headers or the entire messages (in the case of small messages) into a low-latency dedicated buffer. Unfortunately, the time to process long UMQ and PRQ on embedded processors has been reported to be substantially longer than regular host CPU-based implementations, simply because these embedded processors are slower [14, 110, 109]. More importantly, NIC or accelerator memory is usually a very limited resource which can become a scalability barrier when large queues build up [103].

As a purely software solution, hash tables have been proposed to handle the message queue operations [73, 93]. They are however reported to have prohibitive insertion times [110]. The work in [103], even mentioned that hashing can actually have a fairly negative impact on communication latency in almost all situations. Linked lists are used in MPICH [69] and many of its numerous derivatives. Linked lists do not scale over large jobs when lengthy searches occur. An array-based approach is used in Open MPI [79]. While arrays are fast, they are not memory-scalable.

4.2 Motivations

4.2.1 The Omnipresence of Message Queue Processing in MPI Communications

In order for communication to proceed without creating any message queue items, the concept of reception must be totally absent; that is, messages should be able to reach their final destination without any action from the receiving process. In fact, as soon as reception is required, messages will necessarily be queued up because it becomes impossible, without synchronizing before each communication, to make sure that each receive is posted ahead of the arrival of its matching sent message.

The prevalence of message queues in two-sided communications is obvious from a purely semantic point of view. Two-sided communications explicitly necessitate receptions; and synchronization is not always required between the communicating peers. As for collective communications, prior to MPI-3.0, they were necessarily blocking and therefore auto-synchronizing between any two-peer subset of involved processes. As a result, they offered room for limiting or even avoiding message queue buildup. As of MPI-3.0, the introduction of nonblocking collectives has removed the aforementioned auto-synchronization.

The MPI one-sided operations form the only communication model that comes close to avoiding the message queues. In theory, MPI RMA could entirely bypass message queue buildup because of the total absence of reception in its semantics. However, certain MPI libraries such as MVAPICH internally leverage their existing two-sided implementations for one-sided control message transfers; leading to message queue buildup even for MPI RMA.

A trend that is rising message queue prevalence even further is the need to move towards multithreaded MPI and hybrid use of MPI with other paradigms and languages [114] at large scales [3, 8]. An example of one such paradigm is the Intel Threading Building Blocks [46]. Message progression in an MPI process is a global activity; leading to multithreaded MPI processes having centralized message queues shared among all the threads. Therefore, multithreaded MPI communications increase the occurrence of out-of-order message discovery, which increases the probability of a given thread discovering from the network device more messages meant for other threads. Those messages must be put in the UMQ until they are claimed by the right thread.

4.2.2 Performance and Scalability Concerns

MPI performance tuning strategies must be concerned with message queues, which accounted for up to 60% of communication latency in certain tests [109]. While ensuring fast communications is an obvious requirement for MPI, memory consumption is another issue that matters at any scale. At runtime, MPI is viewed as a layer mostly meant to merely transfer the data manipulated by the actual HPC application. As a result, the HPC process does not expect to lose a substantial amount of its available memory to its communication substrate. MPI gains by being lean at large scales to give room to the potentially large amount of application data.

In order to speed up message queue processing, hardware-based approaches have been attempted. Unfortunately, accelerator or NIC processors have been reported to be slower than host CPU [14, 110, 109] for message queue processing. More importantly, hardware solutions are inherently non-scalable. The main issue is resource limits [103]. Unlike host-based approaches which can access a virtually limitless amount of memory, embedded processors are limited to their accessible physical memory which is usually far smaller than host RAM. In situations of embedded memory exhaustion, hardware solutions become a scalability bottleneck. Solutions like embedded memory over-provisioning are expensive; but more importantly, they do not scale throughout generations of system size.

The fastest possible software approach could host the MQIs in a fixed-size array for $O(1)$ accesses. One can trivially notice that resizable arrays suffer performance degradation due to reallocation and data copy. Therefore, any mention of array refers to the C-style fixed-size variant. Array indexing requires a contiguous key whose maximum value is known ahead of its allocation. Out of the message queue search key tuple, only the rank fulfils these conditions. The rank always ranges contiguously from 0 to $n-1$ for any communicator of size n. In comparison, the MPI standard does not specify any starting and end value for contextIds. As for the tag, it bears no contiguity constraint but more importantly, it ranges from 0 to `MPI_TAG_UB`, which can be prohibitively large because it reaches `INT_MAX` in many MPI libraries [79, 71]. The resource issues inherent to allocating an array of two billion slots per communicator per process are obvious; tags are therefore not exploitable. In any case, empirical observations show that message queues grow mostly with job sizes [12, 13, 14, 49], making a rank-based optimization the most promising.

Unfortunately, the once-for-all allocation scheme of arrays corresponds exactly to a linear degradation of memory consumption and is therefore not scalable. This allocation scheme is not even correlated with the actual message queue needs; but with the communicator size. Unless each process in the job exhibits a fully connected communication pattern where each process talks to every other process in the communicator, most indices of the array will never be used. We emphasize that well-crafted MPI programs avoid the fully-connected communication patterns as much as possible to avoid its inherent scalability problems. Instead, most good MPI programs prioritize localized small sub-groups of communications. Furthermore, the current and next generation HPC programs are being more and more demanding in terms of development time and performance tuning. These programs are crafted to solve more complex problems. As a consequence, as much as possible, new HPC programs are composed with existing time-tested, feature-rich and highly tuned domain-specific reusable HPC parallel libraries such as PETSc [83] and Blast [100]. A strongly recommended good practice [41] requires those reusable HPC libraries to duplicate `MPI_COMM_WORLD` in order to isolate their internal communications from the application and other libraries in the same program. As a reminder, `MPI_COMM_WORLD` is a predefined communicator that always exists by default in every MPI job; and its size is the job size. Depending on the number of active parallel libraries, a simple MPI job can therefore have a significant number of job-size communicators; each having at least one contextId. As a result, very large petascale and upcoming exascale MPI jobs will waste a large amount of scarcely used memory if array-based message queues are adopted. However systems growth is generally being accompanied by a certain reduction of the amount of memory per core [3]. It is OK for the overall memory consumption of the whole job to grow linearly with the system size; this is a case of a growing demand coupled with a growing resource. However, it is not practical for the memory consumption per process or per CPU core to grow linearly with the system size. For instance, consider an HPC job of n processes, whose overall dataset memory requirement D is the maximum the system can reasonably host in its aggregated RAM. Assuming that the data is distributed equally over the processes, each process needs an amount of memory $d = D/n$. When both the job size and dataset grow 10 times, each process is still expected to host approximately the same $d = (10D)/(10n)$. The memory consumption pattern of an array-based message queue nullifies this expectation because the available RAM per process gets smaller and smaller than d when the job size grows.

Purely on-demand memory allocation achieves the best memory scalability behaviour. Linked list-based approaches achieve as much as possible this second form of scalability. However, they exhibit a linear degradation with respect to speed of processing. For instance, it is reported in [4] that the linear traversal of a message queue of 4095 items took up to 140 ms on a Blue Gene/P system. Linear searches are all but acceptable at petascale and exascale. With $O(n)$ searches and at fixed CPU core speed, linked list-based MPI message queue searches, and consequently communications, simply get slower on the exact same CPU core when the system size increases. Aggravating matters is the need for next generation supercomputers to become substantially more energy-efficient. Power issues are one of the major challenges mentioned in [8] for the roadmap towards exascale systems. Prominent supercomputer builders such as IBM have even long opted for slower but more energy-efficient CPUs [4]. This trend means that slow queue processing will get even slower in HPC processes doing exactly the same job.

4.2.3 Rethinking Message Queues for Scalability by Leveraging MPI's Very Characteristics

We present a new MPI message queue design which is simultaneously fast and resource-conscious even at large scale. Our approach draws its advantages from a few domain-specific observations. In particular, on top of observing the properties already mentioned for the rank, we noticed that MPI message queue operations follow strictly the following patterns:

1. Search PRQ and Delete MQI if found; otherwise Insert new MQI into UMQ.
2. Search UMQ and Delete MQI if found; otherwise Insert new MQI into PRQ.
3. Search UMQ and Delete MQI if found.
4. Search PRQ and Delete MQI if found.
5. Search UMQ (and simply return search status; do not delete).

The first two scenarios occur for regular communications; including those, such as control message communications, that happen inside MPI without being initiated by the application. They are by far the most common cases. Scenario 3 can happen when `MPI_CANCEL` is called from the sender side. `MPI_CANCEL` is a routine which tries to cancel an already issued communication. When `MPI_CANCEL` is called, control messages meant for the sender are retrieved, if they exist, and discarded. In the new MPI-3.0, scenario 3 can also happen when `MPI_MPROBE` and `MPI_IMPROBE` are invoked. Scenario 4 happens when `MPI_CANCEL` is issued from the receive side. Scenario 5 occurs for `MPI_PROBE` and the new MPI 3.0 `MPI_IPROBE`. The ***probe*** family of MPI routines allow a receive-side process to peek into the MPI middleware in order to check for the arrival of a particular message, without retrieving the message to the application level.

Our design only has to be optimized for the above-listed patterns. We first observe that isolated insertions and deletions never occur in MPI message queue operations. We also observe that searches are part of every single use case. Consequently, a goal of the design strategy is to fundamentally leverage the preceding occurrence of searches as strength for the other operations that it invariably precedes. For instance, the use cases 1 and 2 prompt us to avoid using two separate data frameworks for the PRQ and the UMQ. By hosting them in the same searchable object, a search in the UMQ (use case 2) can serve as a free ride towards the insertion point where a new MQI would be put in the PRQ if the match is not found in the UMQ. The same reasoning applies for use case 1.

Our ***main speed of processing goal*** is to perform localized searches by skipping altogether large portions of the message queue for which the search is guaranteed to yield no result. The new queue design is therefore focused on making the identification of those unfruitful portions easy to detect deterministically without error. To achieve that purpose, we use a multidimensional approach which allows the exploitation of a coordinate mechanism. The coordinate mechanism makes jumps that successively narrow down the search space after the traversal of each dimension. A jump is the mechanism of reaching a coordinate along a dimension. In n dimensions, the rank is transformed in the coordinate $(c_{n-1}, c_{n-2}, ..., c_0)$. We define *dimensionSpan* as the maximum number of distinct coordinate positions on each dimension. We use the same *dimensionSpan* on all the dimensions. The rank is expressed as:

$$rank = \sum_{i=0}^{n-1} c_i \times dimensionSpan^i \tag{4.1}$$

To explain the efficiency of the jump mechanism, let us consider a communicator with 1000 ranks. The ranks range from 0 to 999. A linear search could have to scan all the 1000 ranks before finding an MQI of interest. If dimension is 3; then dimensionSpan is 10. Any rank can be found by scanning at most 10 positions on each dimension; leading to 10 + 10 + 10 = 30 rank traversals in the worst case; instead of 1000 for a vanilla linear linked list.

Our ***secondary speed of processing goal*** is to ensure that any chunk of linear traversal required in the proposed data structure remains very short. In the previous example with 1000 ranks, linear searches never exceed 10. Linear searches, as used in the linked-based design, are actually not unacceptable when they are short; they become harmful only when they are long.

Our ***memory consumption behaviour goal*** is to steer away from any fixed full-provision memory allocation scheme, as used in the array approach, which becomes impractical at large scales. We seek to allocate structural objects to organize hosted MQIs in a search-friendly manner. Those structural objects are not only allocated on-demand but they are used in a way that provides a quick amortization of the memory that they consume.

4.3 A New Message Queue for Large Scales

This section exposes our new scalable message queue design for MPI. In the rest of the section, we use the term *queue* to designate any data structure meant to host MPI message queue items. As a reminder, there is little constraints on contextIds and tags; making them difficult to reason about for efficient resource management. The reasoning which sustains our proposal is mostly built over the rank because 1) it is the principal message queue growth factor [12, 13, 14, 49]; and 2) it bears the already mentioned constraints of contiguity and upper value. For message queue management purposes, we represent a contextId with a *ContextIdLead* object which is attached to its own message queue object.

4.3.1 Reasoning about Dimensionality

High dimensionality tends to favour, in theory, larger speedups for extremely large searches. A concrete example with a communicator of fixed size 10^{16} follows. 10^{16} is not a realistic communicator size; it is purely meant to show very easily the effect of higher dimensions. We choose only power of two dimensions so as to make the computations straightforward. With 2 dimensions, *dimensionSpan* is 10^8 . The worst case traversal time for this 2D structure would be $dimensionSpan + dimensionSpan = 2 \times 10^8$; and the speedup is $10^{16}/\left(2 \times 10^8\right)$. With 4D, the speedup is $10^{16}/\left(4 \times 10^4\right)$; and with 8D, the speedup is $10^{16}/\left(8 \times 10^2\right)$. We notice that the speedup in the worst case increases very fast with the dimensionality. There is however a few problems with using high dimensionality. The first one is that higher dimensionality has a higher fixed search cost. The fixed search cost is the portion of the search latency which must be incurred without regards to where the sought item is located and how sparsely the queue is populated. Because of that fixed cost, high dimensionality can be detrimental for shallow search depths or searches in sparsely-populated queues. When the dimensionality increases, chunks of linear traversals are shorter but there are more of those chunks; and unfortunately, each dimension must always be visited. With each dimension represented as an ordered linked list, the rank 1 for instance is expressed as $0 \times dimensionsSpan^1 + 1 \times dimensionSpan^0$ in 2D; and this expression means that the axis representing $dimensionsSpan^1$ must nevertheless be visited at position 0. In n dimensions, all the axes representing $dimensionsSpan^{n-1}$ to $dimensionsSpan^1$ must still be traversed for rank 1. As a reminder, each dimension ranges from 0 to $dimensionSpan - 1$; meaning that a coordinate of 0 on an axis does not mean that the dimension is skipped during the traversal. Furthermore, how many coordinate objects are deleted or created following insertions or deletions depends on the dimensionality. Actually, each (discrete) coordinate on each dimension is an object. For instance if a rank decomposes into $(c_{n-1}, ..., c_4, c_3, c_2, c_1, c_0)$ in a n-dimension space, its insertion will lead to the creation of an object at position c_i along the axis representing $dimensionSpan^i$ if no MQI having that coordinate for that dimension currently exists in the queue; and this must happen for every dimension. The deletion of an MQI must also trigger the deletion of every coordinate object that the disappearing MQI was the only one to bear. Therefore, we chose 4 dimensions to considerably mitigate these effects on shallow searches, insertions and deletions. As mentioned in Chapter 7, the exploration of dimensions other than 4 is planned for future work.

4.3.2 A Scalable Multidimensional MPI Message Sub-queue

The 4D data-structure described in this section is only meant for large message queues. We split the rank in 4 slices (Figure 4.1(a)). *dimensionSpan* is a power of two in order to allow fast bitwise decompositions. *dimensionSpan* is 4, 8, 16 or 32 for the intervals [1, 256], [257, 4096], [4097, 65536] and [65537, 1048576], respectively. The encoding of *dimensionSpan* takes respectively 2, 3, 4 and 5 bits for each of those intervals. The intervals can keep increasing. In general, a 16-fold maximum communicator size is increasingly covered by adding 1 bit to *dimensionSpan*.

The rank is decomposed in the coordinate quadruple (c_3, c_2, c_1, c_0) such that

$$rank = c_3 \times dimensionSpan^3 + c_2 \times dimensionSpan^2 + c_1 \times dimensionSpan + c_0$$

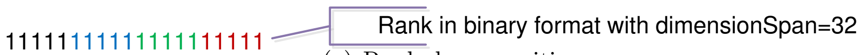
For a communicator of size *size*, *dimensionSpan* is defined by the formula:

$$dimensionSpan = 2^{\lceil log_2(\sqrt[4]{size}) \rceil} \tag{4.2}$$

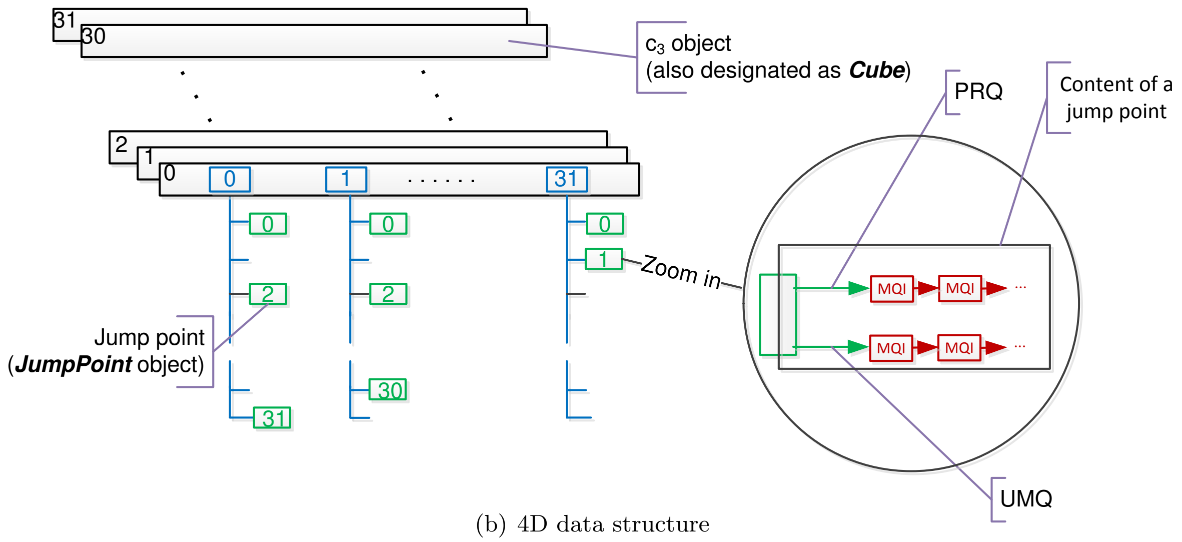
In Equation 4.2, the component inside the `ceil` function tends towards the number of bits of encoding of *dimensionSpan* when the communicator size tends towards the upper limit of the interval covered by *dimensionSpan*. The secondary speed goal is built upon *dimensionSpan*. For any given communicator size, *dimensionSpan* determines the typical length of chunks of linear searches inside our proposed data structure. Since *dimensionSpan* is always reasonably small and grows very slowly when the communicator size increases, linear search lengths are kept under control.

We had two design choices. In the first approach, all 4 coordinate components are used in the jump process to reach a pair of limited length UMQ and PRQ where all the MQIs bear the same rank and only the tag varies. This first approach is beneficial if processes usually maintain several pending messages, resulting in several MQIs having the same rank value and different tag values. Actually, when the same rank can appear numerous times in the MQIs, a linked list for which only the tag varies can already be long enough to justify one such list per rank. If each rank has a very limited number of pending MQIs, a reasonable number of distinct ranks can share the same linked list without creating long and expensive linear traversals. The second approach uses that observation and puts all the MQI whose ranks vary only by the least significant coordinate in the same linked list of UMQ or PRQ. As a reminder, each slice of the rank decomposition can only take on *dimensionSpan* distinct values; meaning that even for a communicator of size 1048576, a maximum of only 32 distinct ranks can be in the aforementioned linked lists.

In this second approach, which we adopt because it is more memory-efficient, the three most significant coordinates c_3 , c_2 , c_1 of the rank participate in the jump process (Figure 4.1(b)); the least significant coordinate is ignored. We define a *Cube* as the object reached by the most significant coordinate c_3, represented in black in Figure 4.1. We define a *JumpPoint* as the object uniquely addressed by the triplet (c_3, c_2, c_1). A JumpPoint is an object which contains a pair of limited length PRQ and UMQ where ranks can vary only by their c_0 coordinates.



(a) Rank decomposition



(b) 4D data structure

Figure 4.1: 4-dimensional sub-queue with $dimensionSpan$ = 32

Speedup and Dimension Representation

For the sake of conciseness, we choose a single *dimensionSpan* (e.g., 32) to explain the rest of the design. With each dimension represented with a linked list, let us consider a search in a queue of 1048576 distinct ranks; each having a single pending message. Even in the worst case, the sought item is found in 32+32+32+32 = 128 scans; leading to a speedup of 1048576/128 = 8192. Such a large speedup is entirely tributary to the dimensional decomposition. If processes have more than a single pending message, the first three dimensions are still searched in a maximum of 32+32+32 scans; the last one could be searched in more than 32 scans. Furthermore, every coordinate on each dimension is allocated only if at least one MQI exists whose decomposition bears that coordinate value on the dimension; leading to a complete on-demand memory allocation scheme.

We can further notice that by representing any of the first three dimensions with an array, its scan time could drop from 32 to 1, leading to further speed improvements; this time with a certain memory trade-off. As a reminder, the last dimension c_0 cannot be represented with an array because it does not participate in the jump process. In particular, using 3 out of the 4 coordinates allows the least significant dimension to represent the same coordinate value more than once in situations where the same rank appears with different tag values. The dimension-wise (not overall) speedup yielded by the array use on any dimension depends on how many distinct coordinates n are usually in use on that dimension. That speedup is actually n. If all the first three dimensions bear the same number of in-use coordinate values, as far as speed is concerned, the dimension to represent with an array does not matter. In particular, if the search must scan the same number n ($n \leq 32$) of positions on each dimension, then representing anyone of the c_3, c_2 or c_1 dimensions with with arrays respectively lead to $(1 + n + n + n)$, $(n + 1 + n + n)$ or $(n + n + 1 + n)$ overall scans.

The likelihood of a dimension having more in-use coordinate values than another one is hard to generalize; it depends on the application and how sparsely or densely its ranks are represented in the queue. It helps to notice that two ranks separated by at least 32^3 falls in two distinct Cubes. If the separation spacing is at least 32^2 without being a multiple of 32^3, the ranks are guaranteed to have two distinct c_2 coordinates. Finally, if the separation is at least 32 without being a multiple of 32^3 or 32^2, then the ranks have two distinct c_1 coordinate values. However, while all 4D queues will always have all their *dimensionSpan* distinct c_2 and c_1 coordinates present, most will usually have less than *dimensionSpan* distinct c_3 values. The number of Cubes present in the 4D data structure is linked to how close the communicator size is to the upper limit of its interval. For instance, with $dimensionSpan = 32$, the smallest communicator size is 65537; and it requires $\lceil 65537/(32^3) \rceil$ distinct values of c_3, which is 3. This means that the communicator is big enough to completely fill Cubes of coordinates $c_3 = 0$ and $c_3 = 1$ and then overflow into the Cube of coordinate $c_3 = 2$. This also means that all the 32 distinct c_2 coordinates and all the 32^2 distinct c_1 objects are in use in Cubes 0 and 1. c_2 and c_1 are thus the two dimensions that could potentially benefit from an array representation due to their frequent occurrences.

The choice of array use on any dimension must also consider the resulting memory degradation impact. Lower dimensions bear a large memory penalty. For instance, there is a total of 32^3 (that is, 32768) c_1 objects and only 32^2 (that is, 1024) c_2 objects for a communicator of 1048576 distinct ranks. Therefore, we elect to represent the c_2 dimension with arrays because it bears the smallest memory penalty. c_3 and c_1 are represented with ordered linked lists. The worst case search can then be performed in 32 + 1 + 32 + 32 = 97 scans; for a speedup of 1048576/97 = 10810. We emphasize that while the use of an array for one of the dimensions yields some additional speedup, the bulk of the improvement is still provided by the break-down into dimensions. In this particular case, the array brings an additional speedup of 10810/8192 = 1.32. In general, when the communicator size (*size*) grows, the speedup S_p associated with just the dimensional decomposition tends towards:

$$\lim_{size\to\infty} S_p = \lim_{size\to\infty} \frac{size}{4 \times dimensionSpan}$$

$$\lim_{size\to\infty} S_p = \lim_{size\to\infty} e^{\left(\frac{3}{4}\times log_2(size) - log_2(4)\right)}$$

$$\lim_{size\to\infty} S_p = \infty \tag{4.3}$$

In comparison, the additional speedup yielded by representing a dimension with an array is asymptotically capped by $S_{p\_ar\_cap}$ such that:

$$S_{p\_ar\_cap} = \lim_{size\to\infty} \frac{4 \times dimensionSpan}{3 \times dimensionSpan + 1} = \frac{4}{3} = 1.33 \tag{4.4}$$

Search Explanation and Example of Early Optimization

To understand how this new message queue achieves localized searches, let us consider an MQI having the rank 80000 in a communicator of 1048576 ranks. The required *dimensionSpan* is 32. 80000 decomposes into 00010—01110—00100—00000; meaning that its (c_3, c_2, c_1) coordinate is (2, 14, 4). The external search of c_3 checks the coordinates along the most significant dimension until it finds the Cube having the coordinate 2. Then, the internal search of the Cube of coordinate 2 first translates into an external search of c_2 object for coordinate 14; and then for an external search of c_1 object for coordinate 4. External searches over ordered coordinates (c_3 and c_1) optimize unfruitful searches. They stop right after the search goes past the sought value and conclude right away that a queue item does not exist. For instance, if coordinate 3 is reached while searching for the Cube; the whole search stops. As for the internal searches, they are performed on at most a single position of each dimension; skipping altogether a large number of irrelevant queue items. In this particular case, $31 \times 32^3 + 31 \times 32^2 + 31 \times 32$ items (that is, 1048544 items) are skipped from the irrelevant 31 Cubes, then the irrelevant 31 c_2 objects of Cube 2 and finally the irrelevant 31 c_1 objects of $(c_3, c_2) = (2, 14)$.

Queueing MPI_ANY_SOURCE

As a reminder, `MPI_ANY_SOURCE` is a wildcard rank which matches all the sender ranks in the same communicator. When an `MPI_ANY_SOURCE` receive searches the UMQ, all the ranks must be probed. Then if a match is not found, because the `MPI_ANY_SOURCE` MQIs do not have a rank that can be decomposed and positioned in the 4D structure, they are positioned in a separate linear sub-queue attached to the ContextIdLead.

4.3.3 Receive Match Ordering Enforcement

The MPI standard requires receivers to match messages coming from the same sender in posted receive order. In order to clarify the ordering constraint, let us designate:

- $recv_k(i)$ as the k^{th} receive call. The message is expected from the sender i.
- $m_l(i)$ as the l^{th} incoming message; and its sender is i.
- $x \leftrightarrow y$ as a match occurring between items x and y.

If a receiver posts $recv_0(0), recv_1(0), recv_2(MPI\_ANY\_SOURCE), recv_3(0)$, then the following examples explain the ordering constraint:

- $m_0(0), m_1(0), m_2(0), m_3(0) \implies$ the set of matches $\{recv_0 \leftrightarrow m_0(0), recv_1 \leftrightarrow m_1(0), recv_2 \leftrightarrow m_2(0), recv_3 \leftrightarrow m_3(0)\}$ occurs.
- $m_0(0), m_1(8), m_2(0), m_3(0) \implies$ the set of matches $\{recv_0 \leftrightarrow m_0(0), recv_2 \leftrightarrow m_1(8), recv_1 \leftrightarrow m_2(0), recv_3 \leftrightarrow m_3(0)\}$ occurs.

Linked list-based queues naturally enforce the receive matching order when they are always inserted into from the tail pointer and always searched from the head pointer. For our 4D proposal, that order is kept as well, as long as there is no `MPI_ANY_SOURCE` item. As a reminder, for the 4D design, queue items of the same ranks are always in the same common short linked lists of the same JumpPoint object. For the general case, we add a monotonically increasing sequence number to the key tuple of all the queue items of the same contextId. Then, each incoming message searches both the ranked PRQ list in the 4D data structure; and the `MPI_ANY_SOURCE` sub-queue if present. If both yield a match, the ordering is enforced by picking the one with smaller sequence number.

4.3.4 Size Threshold Heuristic for Sub-queue Structures

There is a small fixed cost for searching any queue item in the 4D structure; this cost is made of decomposing the rank as well as finding the right Cube and JumpPoint. For insertion, this cost is made of creating a new Cube and/or a new JumpPoint if they were not present before the new queue item could be inserted. Finally, for deletion, a JumpPoint that becomes empty must be removed from the 4D queue; same goes for a Cube that becomes empty. The 4D structure becomes more interesting than the linked list only when searches are (or could potentially be) long enough to overbalance all those small fixed costs. As a reminder, the 4D container is for large jobs; so this constraint is reasonable.

We cannot predict the average search lengths before searches actually happen. However, we always know how big a communicator is at the time of its creation. The communicator size is the maximum number of distinct ranks that the queue can hold; it is also the maximum queue size if each process has a single pending message. Since search lengths are unknown ahead of time, we use those maximum possible queue sizes. We define a threshold below which the sub-queue associated with a ContextIdLead is merely a linked list (Figure 4.2). If the communicator size is above that threshold, the 4D data structure is instead used as sub-queue (Figure 4.2). We define the $PointerTraversalLengthThreshold$ of a contextId as the number of MQIs of distinct ranks after which the last item in a linear sub-queue would incur more pointer traversals than searching any item in the corresponding 4D container. Then we define the threshold as:

$$Threshold = PointerTraversalLengthThreshold \times Adjustment \tag{4.5}$$

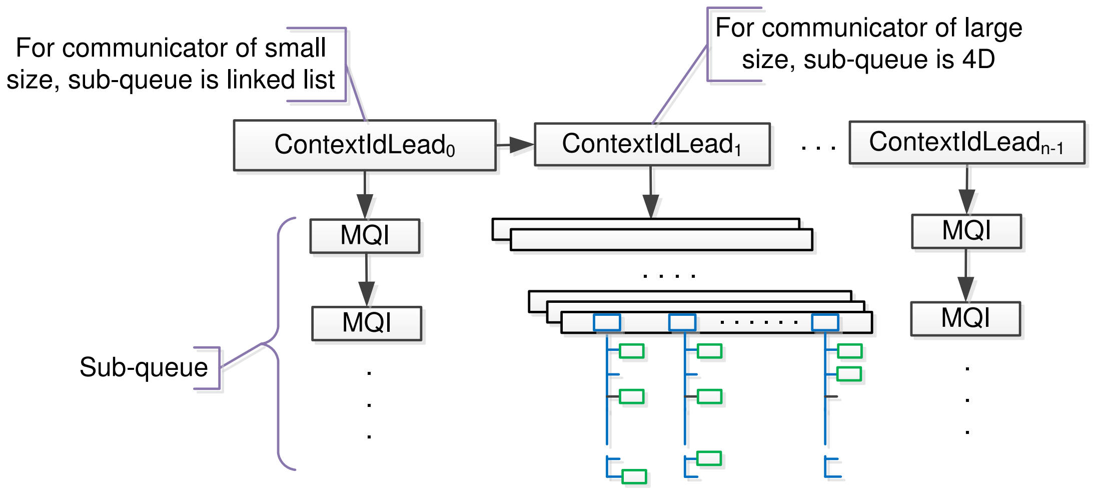

For communicator of small size, sub-queue is linked list
For communicator of large size, sub-queue is 4D
ContextIdLead$_0$
ContextIdLead$_1$
ContextIdLead$_{n-1}$
MQI
MQI
MQI
MQI
Sub-queue

Figure 4.2: New overall MPI message queue design

Adjustment incorporates aspects that are unpredictable such as how often small objects (Cubes, JumpPoints) are created and freed. Furthermore, a linked list traversal is made of a simple loop whose content is quite linear. In comparison, with an equal number of pointer traversals, the 4D structure requires more complex and diverse branchings. Consequently, the threshold cannot be lesser than $PointerTraversalLengthThreshold$. As a result, the minimum value of *Adjustment* must be 1.0.

For all our tests in this chapter, we used an *Adjustment* value of 2.0. The value is empirically based on microbenchmark-level trial-and-error with average message queue search depth being half the message queue size. Concrete values of the threshold are computed as follows. For instance, when the communicator size is in the interval [65537, 1048576] with *dimensionSpan* being 32, searching an item would require in the worst case 32 + 1 + 32 + 32 for Cube, c_2 object, c_1 object and JumpPoint queue scanning. $PointerTraversalLengthThreshold$ is thus 97 and the threshold is 194. That threshold becomes 26, 50 and 98 respectively for communicators whose sizes are respectively in the intervals [1, 256], [257, 4096] and [4097, 65536].

$PointerTraversalLengthThreshold$ is not user-modifiable; it is a characteristic of the 4D structure for each communicator size. $PointerTraversalLengthThreshold$ is computed automatically from the communicator size. *Adjustment* is user-modifiable. However, it incorporates aspects which are difficult to generalize even with a given MPI library. It accounts for variables such as how often the message queue gets empty. Another variable that could influence *Adjustment* could be CPU speed. For instance, a 2.2 GHz Xeon is probably less sensitive to a certain length of linear traversals than an 850 MHz PowerPC. As a result, everything else being equal (including memory access latency, cache miss ratio, object allocation frequency, etc.) the PowerPC gains by using a low *Adjustment* so as to switch to the 4D structure earlier. In general, *Adjustment* is tied to both the specific application and the specific architecture.

Though, it is important to notice that *Adjustment* does not really matter at large scales. Figure 4.3 shows the threshold relation to performance with hypothetical examples. The CPU consumption growth pattern shown in Figure 4.3 does not matter for the purpose of the example; the curves "Slow at scale" (SAS) and "Fast at scale" (FAS) could have been anything but linear. We define S_c , T_a and T_h respectively as the actual communicator size, the accurate threshold value and the heuristic-approximated threshold value. As a reminder, all the thresholds are expressed in communicator size; and T_h is the approximation expressed by Equation 4.5. In Figure 4.3 , lower CPU consumption is better because it means faster queue operation. The aim is to:

- Choose SAS (or the linked list in our particular case) if $S_c < T_a$.
- Choose FAS (or the 4D in our particular case) if $S_c \geq T_a$.

Since T_a is not known, the choice of the sub-queue type depends on T_h. The following scenarios can then happen:

1. $T_h < T_a$:

   Consequences:

   (a) If $S_c < T_h$: The right sub-queue (SAS) is chosen because $S_c < T_a$ is true.

   (b) If $T_h \leq S_c < T_a$: FAS is chosen while SAS should have been chosen.

   (c) If $T_a \leq S_c$: The right sub-queue (FAS) is chosen.

2. $T_h > T_a$:

   Consequences:

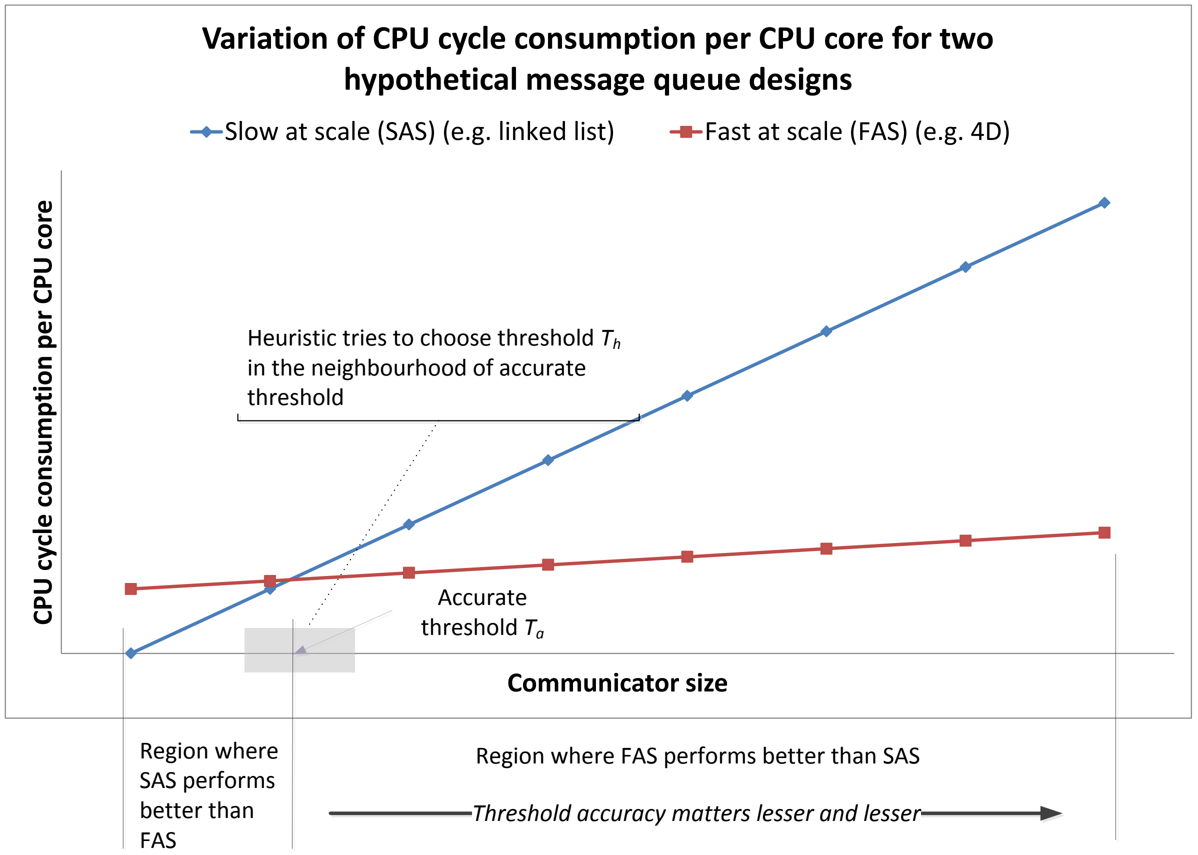


Figure 4.3: Showing threshold accuracy effect on performance

(a) If $S_c < T_a$: the right sub-queue (SAS) is chosen.

(b) If $T_a \leq S_c < T_h$: SAS is chosen while FAS should have been chosen.

(c) If $T_h < S_c$: The right sub-queue (FAS) is chosen.

One can make the following observations:

1. An inaccurate threshold leads to the wrong choice only when S_c falls between T_a and T_h.

2. S_c falling between T_a and T_h usually implies that S_c is still in the neighbourhood of T_a in which case performance variations due to the choice of SAS or FAS can even be assimilated to or balanced out by noise.

3. The extent $|T_a - T_h|$ of the threshold inaccuracy would not matter for large communicators; because S_c will be far beyond both T_a and T_h ; that is, S_c will not fall between

the two threshold values. Threshold inaccuracies will therefore never have any impact for the very use-case for which the 4D is being proposed.

4.4 Performance Analysis

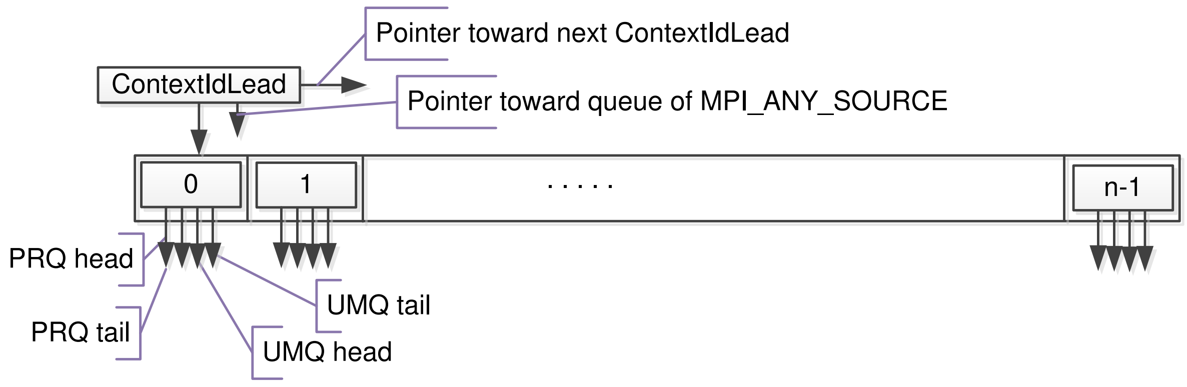
This section analyzes the runtime complexity, memory consumption behaviour and pointer traversal intensiveness for the new proposal. For comparison purposes, we establish the same analyses for an array-based message queue design that we consider as the theoretical optimal approach in terms of speed of processing; and a linked list-based approach that we consider as the optimal design in terms of memory consumption behaviour for the same number of MQIs.

The linked list design is simply made of a pair of linked lists; one for the PRQ and another one for the UMQ. Figure 4.4 shows the array-based design which, because it is built over rank numbers, is a per-communicator approach too. The array-based design resorts to ContextI-dLeads as well. Each ContextIdLead associated with a communicator of size n has an array of n *RankHead* data structures. The RankHead at position i in the array is associated with the MQIs coming from or meant for the process of rank i in the communicator. Each RankHead possesses two pairs of pointers for the head and tails of its UMQ and PRQ. In each of these PRQ and UMQ, only the tag varies. For instance, for a ContextIdLead whose contextId is c, all the MQIs hosted in the PRQ head of RankHead i are such that MQI.contextId $= c$ and MQI.rank $= i$; but MQI.tag can vary. MQIs bearing `MPI_ANY_SOURCE` are hosted in a separate linked list-based sub-queue because their rank field cannot be used as array index. The MPI message ordering constraint is once again enforced with a sequence number approach.

4.4.1 Runtime Complexity Analysis

In this section, we provide a comparative asymptotic analysis of the linked list, the array and our proposed 4D message queue data structures. As stated, the linked list design is the default approach used in MVAPICH [71]. We also implemented the array-based method in MVAPICH for comparison purposes.

ContextIdLeads, Cube and JumpPoint are small objects allocated and freed on demand



Pointer toward next ContextIdLead
ContextIdLead
Pointer toward queue of MPI_ANY_SOURCE
0
1
.....
n-1
PRQ head
PRQ tail
UMQ tail
UMQ head

Figure 4.4: Array-based queue design for a communicator of size n

at the frequency of communicator creation/destruction and queue length changes. Cubes and JumpPoints are the only runtime objects that implement entirely the 4D data structure. There are no other separate runtime objects for each coordinate component.

In all the analysis, we separate search and deletion complexities. Though, we bear in mind that a deletion is always preceded by a search in the very queue which is being deleted from. Furthermore, insertion in any MPI PRQ always happens after the UMQ has been searched in vain and vice versa. This last behaviour allows our proposed complex data structures to offer insertion in $O(1)$ without breaking its structure. For instance, a queue item of rank r about to be inserted in the PRQ means that the Cube and JumpPoint object corresponding to the coordinate decomposition of r have already been reached when searching the UMQ. Even if the Cube or JumpPoint object were not present, their insertion coordinates were at least reached while searching the UMQ.

We designate by:

- k the number of currently active contextIds in the considered process.
- n_j the number of distinct valid ranks in the contextId j.
- t_i the number of queue items associated with the same rank i in any queue of the same contextId. In the UMQ, t_i designates the number of simultaneously pending unexpected messages coming from the process with rank i and meant for the current process. In the PRQ, t_i is the number of posted receives from the current process and meant to match messages coming from the process with rank i.

- a_j the number of `MPI_ANY_SOURCE` queue items associated with contextId j.

Asymptotic Complexities

The Linked List-based Queue: Because the scanning goes through all the t_i queue items of all the n_j ranks of all the k contextIds, searching the linked list happens in:

$$O\left(\sum_{j=0}^{k-1}\sum_{i=0}^{n_j-1} t_i\right) \tag{4.6}$$

This search complexity is essentially made of the Cartesian product of the involved parameters. If there are `MPI_ANY_SOURCE`, the complexity becomes:

$$O\left(\sum_{j=0}^{k-1}\left(\left(\sum_{i=0}^{n_j-1} t_i\right) + a_j\right)\right) \tag{4.7}$$

Both Equation 4.6 and Equation 4.7 are strictly equivalent and express the exact same complexity. It is possible to consider a_j as the t_i associated with the special rank `MPI_ANY_SOURCE`. In that case, `MPI_ANY_SOURCE` is considered as one of the n_j ranks; forcing Equation 4.7 to degenerate to Equation 4.6.

The worst possible search scenario for the linked list approach happens when the sought queue item is at the tail of the queue. It also occurs when the queue item does not exist in the queue; the queue in this case must be traversed entirely just to realize that the search is unfruitful. Deletion always happens at the found position and is therefore:

$$O(1) \tag{4.8}$$

Insertion is $O(1)$ as well because it always happens at the tail of the queue.

The Array-based Queue: As shown in Figure 4.5, a search always requires the traversal of the ContextIdLeads in $O(k)$, the access to the rank slot in the array in $O(1)$ and then the traversal of the queue items associated with the same rank i in $O(t_i)$. The UMQ and PRQ search complexity is thus:

$$O(k + t_i) \tag{4.9}$$

In the data structure shown in Figure 4.5, we assume for each rank slot that the left linked list is the PRQ and the right one is the UMQ.

Whenever an `MPI_ANY_SOURCE` receive is issued, all the UMQ queue items associated with all the ranks must be probed (Figure 4.6). The UMQ search complexity is thus:

$$O\left(k + \sum_{i=0}^{n_j-1} t_i\right) \tag{4.10}$$

As mentioned earlier, `MPI_ANY_SOURCE` queue items build a separate queue of length a_j attached to the ContextIdLead. In that case, for any incoming message, the `MPI_ANY_SOURCE` sub-queue must be searched on top of the linked list associated with the rank of the message

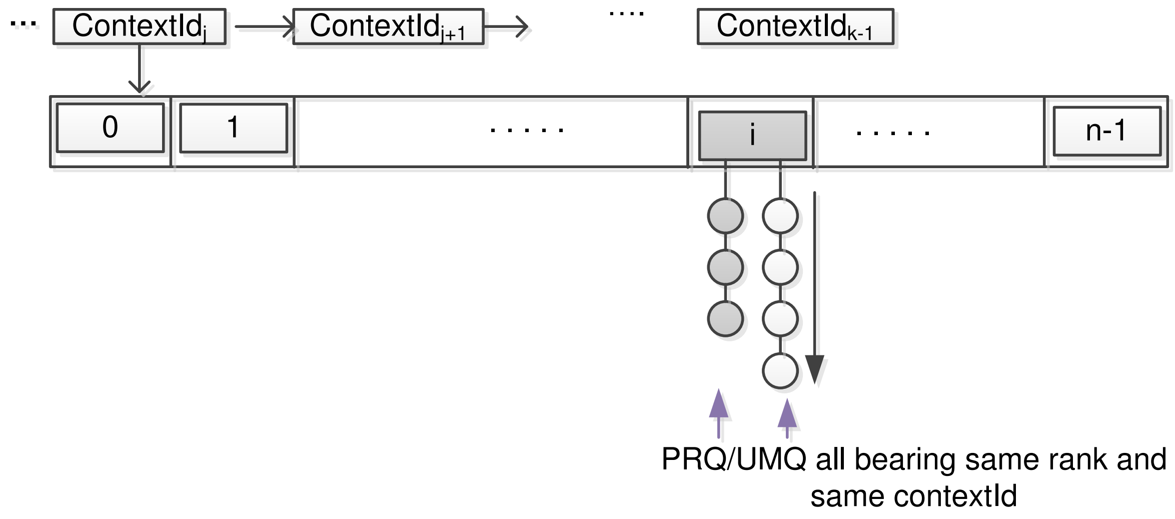


Figure 4.5: Array-based queue search in absence of MPI_ANY_SOURCE

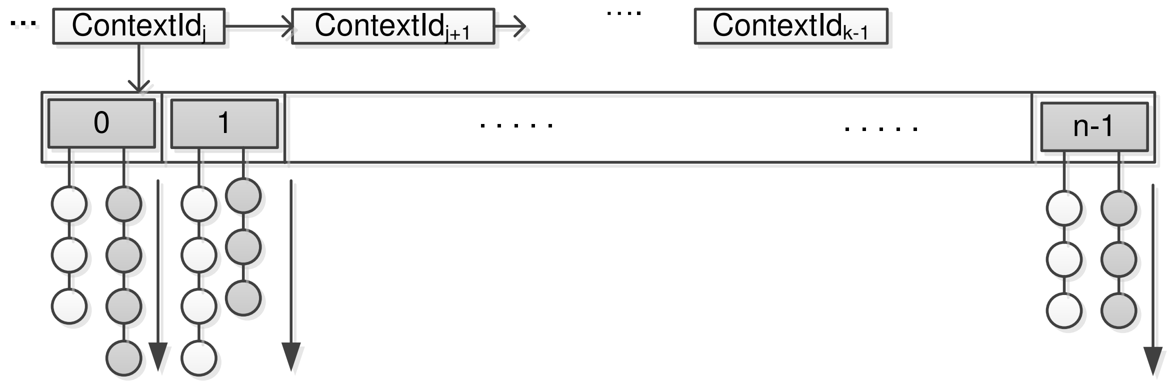


Figure 4.6: Array-based UMQ search for MPI_ANY_SOURCE receive

envelope (Figure 4.7). The PRQ search complexity thus becomes:

$$O\left(k + t_i + a_j\right) \tag{4.11}$$

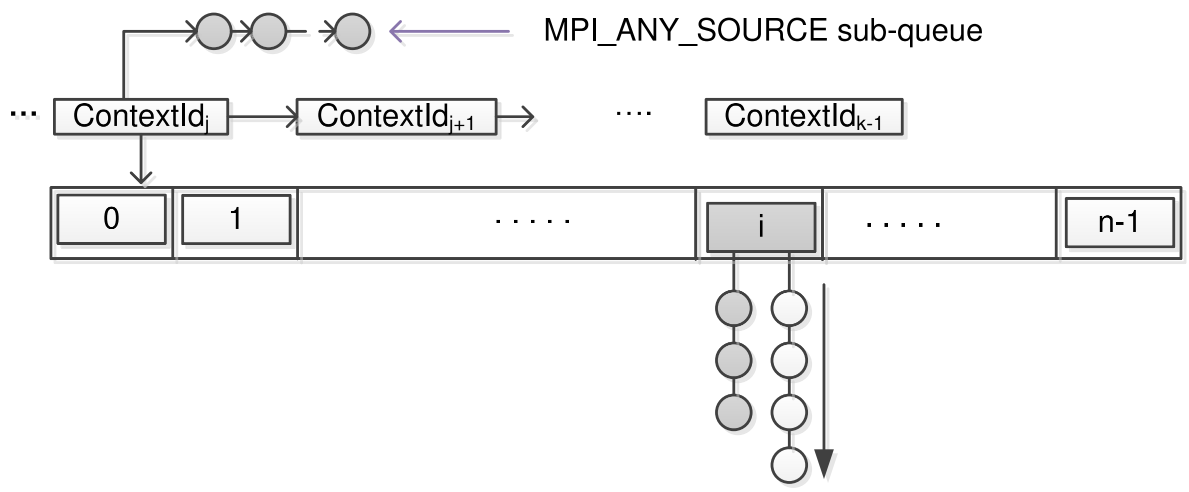


Figure 4.7: Array-based PRQ search when at least one MPI_ANY_SOURCE receive is pending

Deletion always happens at the position where a sought queue item is found. Deletion is therefore $O(1)$. Insertion, no matter where it happens, is $O(1)$ as well because new queue items are simply appended to mere linked lists as shown in Figure 4.8.

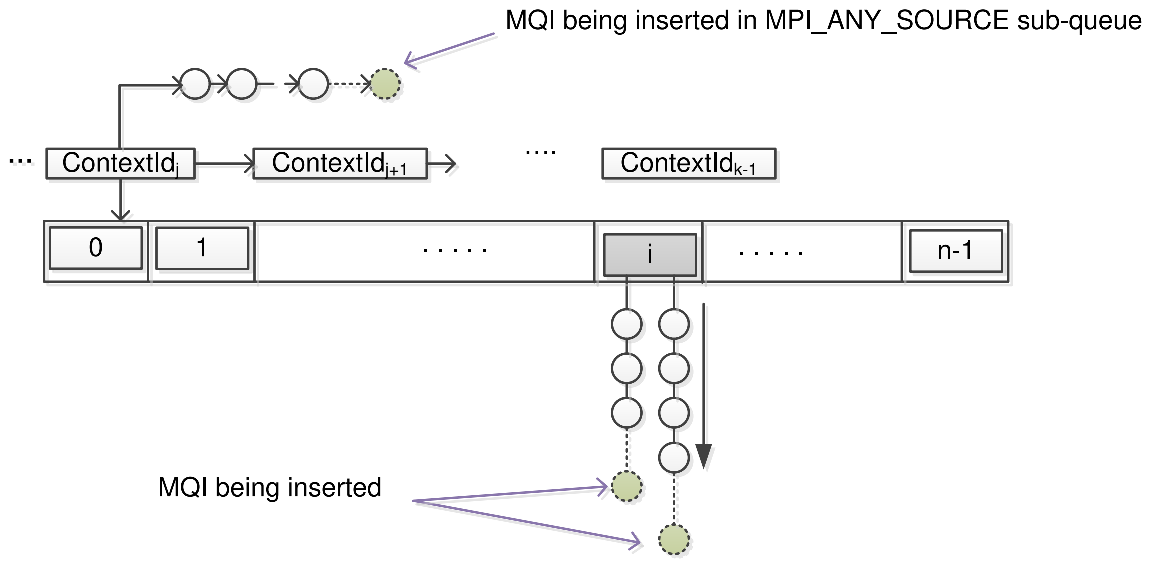


Figure 4.8: Possible insertion points in the array-based structure

The 4D Data Structure Queue: The search procedure in absence of `MPI_ANY_SOURCE` is depicted in Figure 4.9. The right ContextIdLead is found in $O(k)$. Then the rank is decomposed in four slices of *dimensionSpan* each. As a reminder (Equation 4.2), $dimensionSpan = 2^{\lceil log_2(\sqrt[4]{n_j})\rceil}$; with n_j being the size of the communicator associated with the contextId j. The right Cube is found in $O\left(2^{\lceil log_2(\sqrt[4]{n_j})\rceil}\right)$ (Figure 4.9(a)). Then, the right c_2 coordinate is found in $O(1)$ because it is an array index in the Cube (Figure 4.9(b)). The c_1 coordinate is found in $O\left(2^{\lceil log_2(\sqrt[4]{n_j})\rceil}\right)$ (Figure 4.9(c)); and the JumpPoint is reached. For both the PRQ and the UMQ, the JumpPoint contains a maximum of *dimensionSpan* distinct ranks as well; with the rank i having t_i queue items (Figure 4.9(d)). The JumpPoint traversal therefore happens in $O\left(\sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right)$. The overall search complexity for both the UMQ and the PRQ is thus:

$$O\left(k + 2^{\lceil log_2(\sqrt[4]{n_j})\rceil} + 1 + 2^{\lceil log_2(\sqrt[4]{n_j})\rceil} + \sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right)$$

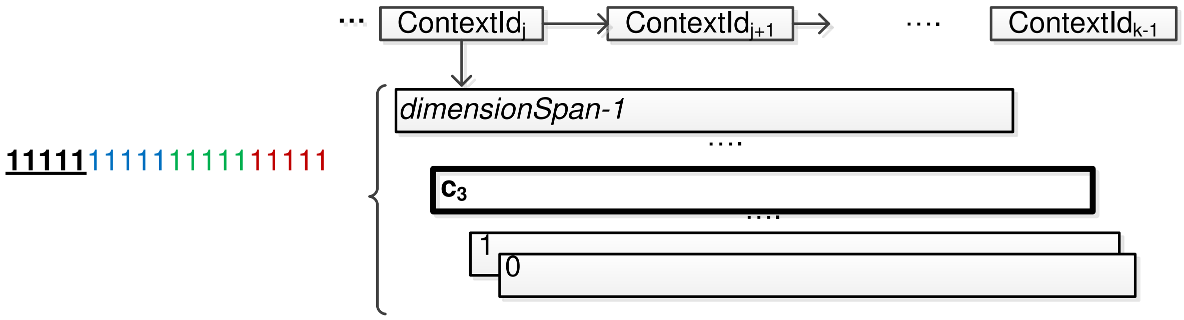
or simply

$$O\left(k + \sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right) \tag{4.12}$$

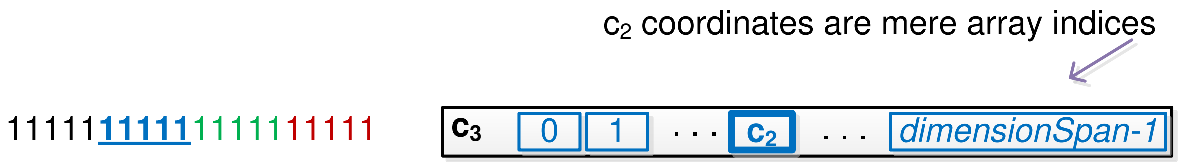
When a posted receive bears `MPI_ANY_SOURCE`, once the ContextIdLead is found, the UMQ of all existing JumpPoints must be probed. This corresponds to searching all the c_2 and c_1 coordinates of all the Cubes. While the complexity of that UMQ search for `MPI_ANY_SOURCE` can be derived from Figure 4.9 as well, one can notice that searching all the UMQs of the data structure comes down to searching all the t_i UMQ items of all the n_j ranks in the communicator. Such a search simply has a complexity $O\left(\sum_{i=0}^{n_j-1} t_i\right)$. The UMQ search complexity when an `MPI_ANY_SOURCE` receive is posted is thus $O\left(k + \sum_{i=0}^{n_j-1} t_i\right)$; which is the same as Equation 4.10 for the array-based data structure.

Finally, when at least one `MPI_ANY_SOURCE` queue item is pending, the PRQ search performs all the operations depicted in Figure 4.9 and then searches the existing `MPI_ANY_SOURCE` sub-queue in $O(a_j)$ (Figure 4.10). The PRQ search complexity in that case thus becomes:

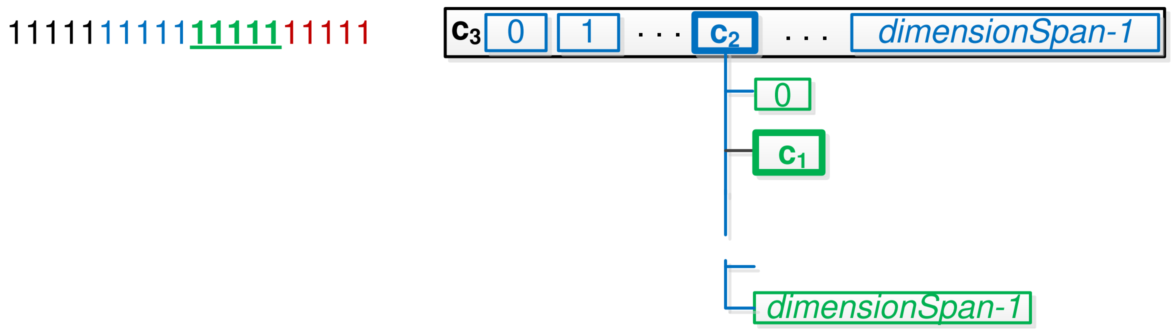
$$O\left(k + \left(\sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right) + a_j\right) \tag{4.13}$$



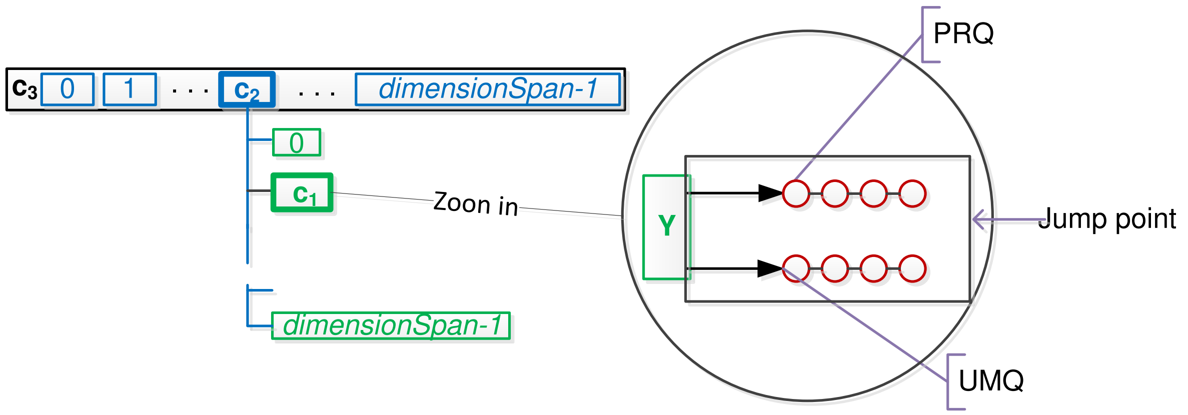
(a) Reaching the right c_3 coordinate



(b) Reaching the right c_2 coordinate



(c) Reaching the right c_1 coordinate



(d) Finding the MQI in limited linked lists of JumpPoint object

Figure 4.9: 4D structure queue search in absence of MPI_ANY_SOURCE

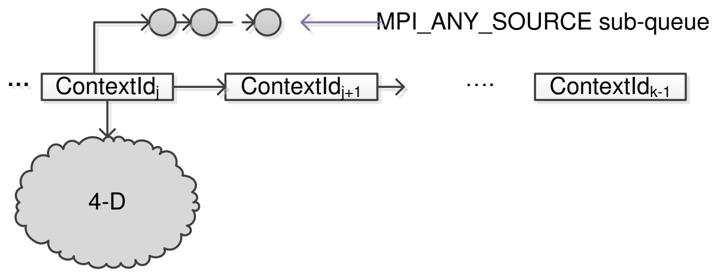


Figure 4.10: 4D structure PRQ search in presence of MPI_ANY_SOURCE

Deletion happens at the point where the queue item is found and is therefore $O(1)$. Insertion of a new queue item is $O(1)$ as well, and can be explained as follows (Figure 4.11). `MPI_ANY_SOURCE` queue items are simply appended to the end of the `MPI_ANY_SOURCE` sub-queue. If the queue item has a specific rank instead, its insertion always happens at the point where the previous search failed. The previous unfruitful search failed because the rank decomposition showed that either 1) the sought Cube is inexistent; 2) the Cube exists but the JumpPoint object is inexistent because the c_2 or c_1 coordinates yielded by the decompositions are currently unused; or 3) both the Cube and JumpPoint exist but the queue item does not. In the first case, a new Cube and JumpPoint object are created and inserted before the queue item. In the second, the JumpPoint object is inserted and then the queue item is. In the third case, the queue item is appended right away to the right PRQ or UMQ of the JumpPoint where the search failed. Each of the aforementioned operations is $O(1)$.

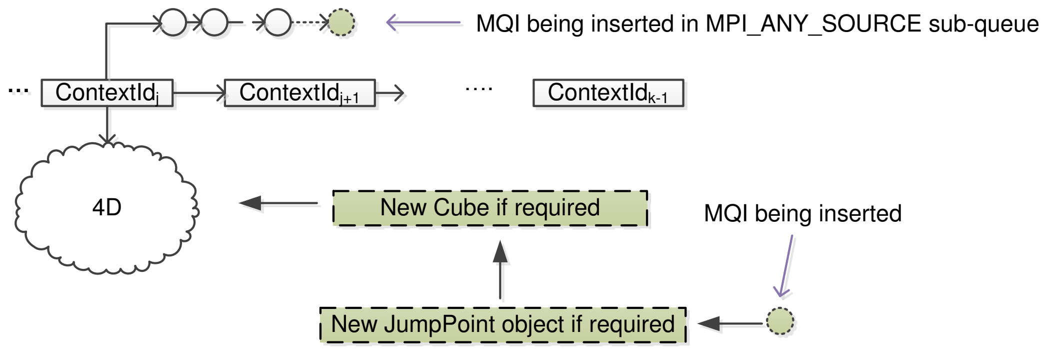


Figure 4.11: Possible insertion points in the 4D structure

Runtime Complexity Analysis Summary: The runtime complexities are summarized in Table 4.1. A few observations stem from the complexities:

1. When there are more than a single contextId (i.e., $k > 1$), the runtime search complexity of the linked list design has a cubic behaviour in terms of the used parameters (Equation 4.6, Equation 4.7). In comparison, the array-based approach and the 4D proposal remain entirely quadratic without regard to the presence of `MPI_ANY_SOURCE` (Equation 4.9 – Equation 4.13).

2. When there is no `MPI_ANY_SOURCE`, the parameter n_j which usually impacts the queue operation times the most is absent from the array-based approach (Equation 4.9) for both the UMQ and the PRQ. In the 4D approach, this parameter has a logarithmic behaviour (Equation 4.12), meaning that its impact on the cost grows more and more slowly when scale increases.

3. When there are `MPI_ANY_SOURCE`, the PRQ search in the linked list must potentially scan all the `MPI_ANY_SOURCE` queue items of all the existing contextIds (Equation 4.7). In comparison, the PRQ search of the array (Equation 4.11) and the 4D structure (Equation 4.13) grows only by the number of `MPI_ANY_SOURCE` queue items pending for the sole contextId of interest.

4. More generally, as shown by the parameter k for searches, each new communicator adds a single pointer visitation to the array (Equation 4.9 – Equation 4.11) and the 4D structure (Equation 4.10, Equation 4.12, Equation 4.13) while it adds the quadratic term $\sum_{i=0}^{n_j-1} t_i$ or $\left(\sum_{i=0}^{n_j-1} t_i\right) + a_j$ to the linked list (Equation 4.6, Equation 4.7).

Finally, none of the complexities is impacted by the use of `MPI_ANY_TAG`.

Intuitive Grasp of Runtime Costs

Asymptotic complexities do not always allow an intuitive grasp of comparative costs. We therefore propose to quickly pass the three approaches through a test performed in [4]. The original test, based on 4096 processes, searches for an item which is the last one in a queue of 4095 items. We use a variant of the test where each process has a single queue item in the queue. Assuming a single communicator, the number of pointer operations required for the search is shown in Table 4.2 for each of the message queue architectures.

Table 4.1: Runtime complexity summary

Linked list	PRQ/UMQ search without MPI_ANY_SOURCE	$O\left(\sum_{j=0}^{k-1}\sum_{i=0}^{n_j-1} t_i\right)$	Eq. 4.6
	PRQ/UMQ search with MPI_ANY_SOURCE	$O\left(\sum_{j=0}^{k-1}\left(\left(\sum_{i=0}^{n_j-1} t_i\right)+a_j\right)\right)$	Eq. 4.7
	Insertion	$O(1)$	Eq. 4.8
	Deletion	$O(1)$	Eq. 4.8
Array	PRQ/UMQ search without MPI_ANY_SOURCE	$O(k+t_i)$	Eq. 4.9
	UMQ search with MPI_ANY_SOURCE	$O\left(k+\sum_{i=0}^{n_j-1} t_i\right)$	Eq. 4.10
	PRQ search with MPI_ANY_SOURCE	$O\left(k+t_i+a_j\right)$	Eq. 4.11
	Insertion	$O(1)$	Eq. 4.8
	Deletion	$O(1)$	Eq. 4.8
4D	PRQ/UMQ search without MPI_ANY_SOURCE	$O\left(k+\sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right)$	Eq. 4.12
	UMQ search with MPI_ANY_SOURCE	$O\left(k+\sum_{i=0}^{n_j-1} t_i\right)$	Eq. 4.10
	PRQ search with MPI_ANY_SOURCE	$O\left(k+\left(\sum_{i=0}^{2^{\lceil log_2(\sqrt[4]{n_j})\rceil}-1} t_i\right)+a_j\right)$	Eq. 4.13
	Insertion	$O(1)$	Eq. 4.8
	Deletion	$O(1)$	Eq. 4.8

In the linked list case the search would always scan through all the items. In the array-based case, the search always scans through three pointers; one for the ContextIdLead, one for the position of the rank by indexing the array, and the last one for the queue item being looked for. In the 4D structure, the search scans through one pointer to find the ContextIdLead, *dimensionSpan* pointers to find the Cube, one pointer to reach the c_2 coordinate, *dimensionSpan* pointers to find the c_1 coordinate and *dimensionSpan* pointers to reach the queue item in the JumpPoint. The goal of this analysis is to show intuitively how slowly the 4D approach degrades compared to the linked list when scales grow. The array-based approach is communicator size-agnostic.

Table 4.3 shows the same test for 10 communicators, each having the same size. In this case, the number of pointer operations for both the array-based approach and the 4D proposal grows exactly by the number of additional communicators. In comparison, the number of pointer operations for the linked list-based approach grows by the sum of all the items associated with each of these communicators.

Table 4.2: Number of pointer operations required to reach the last queue item for different communicator sizes (case of a single communicator)

Communicator size	Linked list	Array	4D
4096	4095	3	26
65536	65535	3	50
1048576	1048575	3	98

Table 4.3: Number of pointer operations required to reach the last queue item for different communicator sizes (case of multiple communicators)

Communicator size	Linked list	Array	4D
4096	$9 \times 4095 + 4095$	$9 + 3$	$9 + 26$
65536	$9 \times 65535 + 65535$	$9 + 3$	$9 + 50$
1048576	$9 \times 1048575 + 1048575$	$9 + 3$	$9 + 98$

4.4.2 Memory Overhead Analysis

The memory overhead of the proposed 4D approach is difficult to model concisely in analytic form because it increases by steps, according to *dimensionSpan* and according to the distribution of the ranks in the queues. In fact the memory overhead of the 4D design has a deterministic trend only when the ranks are represented in a contiguous manner; or when all the ranks have at least one item in the queue.

The memory overhead compared to the linked list is computed as how much more memory is required by the array-based method and the proposed 4D structure to host the same number of queue items. It is important to mention that only the number of distinct ranks potentially impact the memory overhead for both the array and the 4D approach. As a reminder, both the array and the 4D are built over the rank; not the tag. As a result, the memory overhead behaviour is easier to compute only when each process has a single pending message; so as to avoid the same rank showing up more than once. In fact the same rank appearing twice does not increase the number of RankHeads, Cubes or JumpPoints. Consequently, when the same rank appears more than once, the computation has to account for it only once to be accurate.

We designate by n_{MQI} the number of MQIs. For a communicator of size S, we proceed to compute the memory consumption M_{ll} , M_{ar} and M_{4D} of the linked list-based, array-based and 4D designs respectively. The linked list design only uses chains of MQIs; thus:

$$M_{ll} = n_{MQI} \times sizeof(MQI) \tag{4.14}$$

The array design (see Figure 4.4) is made of the ContextIdLead object pointing to an array of S RankHead objects. Thus:

$$\begin{aligned} M_{ar} =& sizeof(ContextIdLead) + S \times sizeof(RankHead)+ \\ & n_{MQI} \times sizeof(MQI) \end{aligned} \tag{4.15}$$

The 4D design (Figure 4.1 and Figure 4.2) is made of the ContextIdLead object, the number n_{JP} of JumpPoint objects required to host all the MQIs, and the number n_{c3} of Cubes required to host all the n_{JP} JumpPoint objects. As a reminder, c_2 and c_1 are actually not runtime objects; they are tied to the Cube and JumpPoint objects. Thus:

$$\begin{aligned} M_{4D} =& sizeof(ContextIdLead) + n_{c3} \times sizeof(Cube)+ \\ & n_{JP} \times sizeof(JumpPoint) + n_{MQI} \times sizeof(MQI) \end{aligned} \tag{4.16}$$

A JumpPoint can contain up to *dimensionSpan* distinct MQIs; thus, for contiguously distributed ranks:

$$n_{JP} = \lceil n_{MQI}/dimensionSpan \rceil \tag{4.17}$$

JumpPoint objects have a fixed size because they simply host pointers to linked lists of PRQ and UMQ. A Cube can contain up to *dimensionSpan*^{3} distinct MQIs; thus, for contiguously distributed ranks:

$$n_{c3} = \lceil n_{MQI}/dimensionSpan^3 \rceil \tag{4.18}$$

For the same n_{MQI}, the respective memory overheads O_{ar} and O_{4D} of the array and 4D designs compared to the linked list approach are expressed as follows:

$$O_{ar} = M_{ar} - M_{ll}$$

$$O_{ar} = sizeof(ContextIdLead) + S \times sizeof(RankHead) \tag{4.19}$$

$$O_{4D} = M_{4D} - M_{ll}$$

$$\begin{aligned} O_{4D} =& sizeof(ContextIdLead) + n_{c3} \times sizeof(Cube)+ \\ & n_{JP} \times sizeof(JumpPoint) \end{aligned} \tag{4.20}$$

Since the c_2 dimension hosted in Cubes is an array of *dimensionSpan* slots, Cubes are entities whose sizes depend on *dimensionSpan* (see Figure 4.1). We designate by Cube(d) the

Table 4.4: Runtime object sizes

Object	Size in bytes
ContextIdLead of array	56
RankHead	32
ContextIdLead of 4D	56
Cube(8)	80
Cube(16)	144
Cube(32)	272
JumpPoint	48

programmatic Cube associated with $dimensionSpan = d$. The runtime sizes of the objects on a 64-bit Linux system are showed in Table 4.4.

Table 4.5 shows the values of O_{ar} and O_{4D} for communicators of different sizes. For each size, the results are presented for a few different values n_{MQI} of the number of queue items. We first observe that the array-based approach can waste a lot of memory compared to the 4D approach when there are a few MQIs in the queues. An example of such a situation occurs when there are only 1000 queue items in a 1048576-rank communicator queue. The tests also show two interesting behaviours for our 4D method. First, the memory overhead increase is sub-linear in terms of the number of queue items. This behaviour can be observed at a fixed communicator size. Second, its increase rate tends to flatten considerably when the communicator size grows. The memory overhead growth rate is smaller for the communicator of size 65536 compared to the communicator of size 4096. That growth is even smaller for the communicator of size 1048576. The two behaviours are very consistent with the scalable comportment we expect to provide. The case when all the queue items of the communicator are represented, the test labelled "full", is meant to show the maximum memory overhead for the 4D design. Even in that case, the 4D approach is 5.19 times lighter on memory than the array-based design for a 4096-rank communicator. This ratio becomes 10.54 and 21.19 for 65536 and 1048576 communicator sizes respectively.

We must point out that the previous test assumed contiguous ranks. The 4D mechanism does not behave similarly regarding memory consumption when the ranks are sparsely distributed. In particular, two queue items whose ranks are separated by at least $dimensionSpan^3$ are located in two distinct Cubes; meaning that the maximum number of Cubes of the container

Table 4.5: Comparative memory overhead of the array-based approach and the 4D approach

Communicator size (S)	Number of MQIs (n_{MQI})	Array overhead (O_{ar})	4D overhead (O_{4D})
4096	1	128.05 KB	184 B
	100	128.05 KB	760 B
	1000	128.05 KB	6.07 KB
	4096(i.e., full)	128.05 KB	24.68 KB
65536	1	2.00 MB	248 B
	100	2.00 MB	536 B
	1000	2.00 MB	3.15 KB
	65536(i.e., full)	2.00 MB	194.30 KB
1048576	1	32.00 MB	376 B
	100	32.00 MB	520 B
	1000	32.00 MB	1.82 KB
	1048576(i.e., full)	32.00 MB	1.51 MB

can be required for no more than *dimensionSpan* distinct ranks. Similarly, two ranks separated by at least *dimensionSpan* will require two distinct JumpPoints. The maximum memory overhead is reached with $dimensionSpan^3$ ranks each separated by *dimensionSpan*. When that happens, all the JumpPoints are allocated in the 4D container; this means implicitly that all the Cubes are allocated as well to host those JumpPoints. However, once that maximum is reached, the overhead stops increasing when the other $(dimensionSpan - 1) \times dimensionSpan^3$ additional distinct ranks are added. For instance, with a communicator of size 1048576, the memory overhead of the 4D container reaches 1.51 MB for no more than $32^3 = 32768$ distinct ranks if these ranks are distributed in the fashion 0, 32, 64, 96, . . . , 1048544. However, once that maximum is reached, the memory overhead stops increasing for all the other $31 \times 32^3 = 1,015,808$ distinct ranks.

4.4.3 Cache Behaviour

Cache behaviour can generally depend on locality of access. However, there is little guarantee of any such locality occurring in any of the three message queue architectures. For the 4D data structure, even though Cube and JumpPoint objects are allocated in pools where objects of the same nature (Cube or Jumpoint) occupy contiguous virtual memories, there is no guarantee for two Cubes or two JumpPoints being adjacent when they are actually used in a 4D queue. Arrays are relatively cache-friendly; however, there is no potential for locality exploitation for a given search because finding the rank in the array occurs in a single jump.

In fact, it is difficult to accurately model the message queue cache impact because the

HPC job does other data accesses for its own application data and for other middleware-level operations that are not necessarily queue-related. However, everything else kept equal, the absolute number of cache misses in general depends on the number of memory accesses. A certain level of cache performance prediction can therefore be made if an application execution is known to perform intensive message queue activities. The number of queue-related memory accesses tends to be proportional to the number of pointer scans (Table 4.4 and Table 4.5). In fact, each pointer traversal is potentially a cache miss; especially when the previous pointer is not adjacent in terms of virtual address. Then, dereferencing each pointer to check for its contained data is potentially another cache miss. As the linked list message queue does substantially more memory accesses (pointer scans) for long enough queue searches, it is expected to perform much less than the array and 4D message queues for the same MPI application. The difference of cache performance between the array and the 4D might not be easily noticeable in practice as shown by their small gaps in terms of pointer scans (Table 4.4 and Table 4.5).

4.5 Practical Justification of a New Message Queue Data Structure

We first remind that hash tables have not been envisioned because they have been proposed before [73, 93] and deemed prohibitively slow for insertions [110]. There are several other existing data structures that allow fast searches. In particular, the large family of tree-based data structures were initially considered as candidates. We did a quite thorough analysis of several of them before considering a totally new container. Non self-balancing trees suffer severe degradations and are not considered in our analysis. While self-balancing trees offer very good performance, they all render insertion and deletion less trivial, for the sake of the balancing that yield the good search performance.

For the sake of justification, we compare how red-black trees, which are a special case of a B-tree, perform compared to our 4D data structure. The choice of red-black trees resides in their well-known performance that justifies their popularity. For instance, they are heavily used in the Linux kernel for schedulers, the high-resolution timer, the ext3 filesystem and the virtual memory areas tracking [37]; just to name a few examples. We implement both the 4D and the red-black tree-based message queues offline (that is, not inside MPI). Then, we test both designs by reproducing the message queue activity patterns found in MPI (see Section 4.2.3). We do not reproduce pattern 2 because it is similar to pattern 1; and we do not reproduce pattern 4 due to its similarity with pattern 3. No `MPI_ANY_SOURCE` is used because it is not a differentiator between the two designs. In order to encompass all the situations where an approach might have a distinctive advantage over the other one, we generate the following rank insertion and deletion patterns:

- Insert in increasing rank order; remove in increasing rank order.
- Insert in increasing rank order; remove in decreasing rank order.
- Insert in decreasing rank order; remove in decreasing rank order.
- Insert in decreasing rank order; remove in increasing rank order.
- Insert iteratively from both ends and remove in the same order. For instance, for 256 ranks, we insert 0, 255, 1, 254, 2, 253,. . . , 127, 128; and we remove in the same sequence.
- Insert iteratively from both ends and remove in the reverse order. For instance, for 256 ranks, we insert 0, 255, 1, 254, 2, 253. . . , 127, 128; and we remove in the sequence 128, 127, 129, 126,. . . , 255, 0.

The tests are performed on a node having two quad-core 2 GHz AMD Opteron 2350 processors and 8 GB of RAM. The binaries are created with GNU GCC 4.4.4 and run on an x86_64 Linux kernel 2.6.32. We realize that for both designs, the performance is not impacted by the data insertion and deletion sequence. Consequently, we report only the outcome of the first sequence in Table 4.6. A different binary is generated for memory tests to avoid instrumentation impacts on the latency results. Each test is averaged over 3 iterations. For each latency test, we perform n insertions, followed by n deletions or searches depending on the pattern; with n being the number of ranks. The tests are performed for one and five messages per rank to simulate the cases of single and multiple tags respectively. All the tests are performed for a single contextId because contextIds are not a differentiator between the two approaches. In fact, contextIds are searched before any of the queues (4D or tree) is reached.

We can notice that the 4D design is always better than the tree design. More importantly, Table 4.7 shows that the 4D design is far more memory-efficient than the tree. The memory overhead of the 4D container is computed at runtime with Equation 4.20. However, since the ranks are not necessarily linearly inserted in the queue, the numbers n_{c3} and n_{JP} of Cubes and JumpPoints are not computed with Equation 4.18 and Equation 4.17; they are rigorously tallied in real-time during execution by means of simple incrementation/decrementation upon allocation/deallocation. With the same reasoning of Section 4.4.2, the memory overhead of the tree is computed as how much more memory it adds, compared to the linked list, to hold the same number n_{MQI} or queue items. Its final formula is as follows:

$$O_{rb\_t} = sizeof(EmptyTreeObject) + n_{MQI} \times sizeof(TreeNode) \tag{4.21}$$

For reasons already mentioned in Section 4.4.2, the memory overhead tests are performed for a single message per rank. Trees are better than arrays because they allocate memory only on demand; meaning that they would not waste large amounts of fixed memory for large communicators which are sparsely used with respect to message queue. However, they do generate an amount of memory overhead strictly proportional to the number of distinct ranks in the queue. Furthermore, no matter the tree design, simply for the existence of left, right and possibly parent pointers, tree nodes are more expensive memory-wise than each RankHead object used for the array. Consequently, when all the ranks are represented in the message queue, trees, without regards to their particular design, are even more expensive than the array-based approach.

It is important to mention that we did make a design choice for the tree design. Each rank is represented with a tree node; yielding the fastest possible tree-based representation of the message queue. We could have decomposed the rank; and used only a certain number of bits as the tree key. The other ignored bits would lead to linear traversals; as we do with the c_0 coordinate inside JumpPoint objects in the 4D design. This approach would have resulted

Table 4.6: Comparative speed test (in seconds) between the 4D design and a red-black tree design

		4D		Tree	
		1 msg	5 msgs	1 msg	5 msgs
256 ranks	Pattern 1	4.95E-04	1.28E-02	2.78E-02	4.00E-02
	Pattern 3	5.15E-04	1.20E-02	2.51E-02	3.85E-02
	Pattern 5	5.39E-04	1.22E-02	1.45E-03	1.33E-02
16,384 ranks	Pattern 1	1.64E+00	8.71E+01	4.85E+00	9.05E+01
	Pattern 3	1.62E+00	8.73E+01	4.70E+00	9.05E+01
	Pattern 5	1.62E+00	8.70E+01	3.07E+00	8.90E+01
131,072 ranks	Pattern 1	2.33E+02	5.99E+03	4.62E+02	6.21E+03
	Pattern 3	2.33E+02	5.98E+03	3.40E+02	6.09E+03
	Pattern 5	2.33E+02	5.99E+03	4.66E+02	6.22E+03

Table 4.7: Memory overhead (in KB) of the 4D and red-black tree designs compared to the linked list design (1 message per rank)

	4D		Tree	
	Peak	Average	Peak	Average
256 ranks	3.242188	1.695312	14.05469	7.082031
16,384 ranks	48.61719	24.42969	896.0547	448.082
131,072 ranks	193.117188	96.742188	7168.055	3584.082031

in less memory consumption but would have slowed down the tree. The usual strength of binary search trees resides in their speed; and we wanted to compare the 4D design against that strength at its best. In any case, we claim the superiority of the 4D design in terms of memory consumption degradation even if only part of the rank bits is used as tree key.

The 4D data structure makes jumps to skip many MQI having ranks that are guaranteed not to be the sought one. These jumps bring in mind skip lits [112]; another search data structures based on jumps. In fact, after showing that the 4D approach outperforms red-black trees, it is technically redundant to present a full comparison with skip lists. The advantage of skip lists over balanced binary search trees such as red-black and AVL is the simplicity of the data structure both at conceptual level and at implementation time [70]. While red-black trees deterministically have $O(log_2(n))$ runtime complexity, skip lists ***probabilistically*** have $O(log_2(n))$ runtime complexity. In the worst case, a quite unlikely scenario, the runtime complexity of a skip list is simply $O(n)$ because it degenerates to a doubly-linked list. As stated in [70], skip lists do not provide a firm logarithmic upper-bound guarantee; that guarantee, if required, should be sought with binary search trees such as red-black and AVL. It is possible as shown in [70] to have skip lists with a deterministic structure and speed behaviour; with added costs in memory consumption as well as insertion/deletion complexity.

Skip lists have layers of doubly-linked lists. The lowest level, where the actual data can reside is made of leaf nodes. The upper levels are meant for jumps only and do not contain any useful data. Each level is an ordered doubly-linked list; with each node of every upper level additionally pointing down to a node of a lower level so as to allow the jumps that characterize skip lists. The worst case in terms of speed for the skip list is the best case in terms of memory; and that case occurs when only the leaf level exists. Even in that best case of memory consumption, a skip list is more expensive than an array when all the ranks of a communicator are represented. In general and without respect to the structure it ends up having, as long as it does not degenerates to a doubly-linked list of leaf nodes only, a skip list is always more expensive than the equivalent red-black tree in terms of memory consumption. In the red-black tree, every node at the exception of the root (possibly) can be used to store short linked lists of UMQ and PRQ. In the skip list, only the leaf nodes can be used for that purpose. With a skip list whose structure approaches the structure of a balanced binary search tree, $O(nlog_2(n))$ nodes is required. In comparison, a balanced binary search tree such as red-black requires no more than n or $n + 1$ nodes to achieve the same goal.

4.6 Experimental Evaluation

The experimental setup is an 11-node InfiniBand cluster. Each node is a quad-socket AMD Opteron 6276 (Interlagos), having a total of 64 processor cores and 128 GB of memory. Therefore, the cluster has a total of $64 \times 11 = 704$ processor cores and 11×128 GB $= 1.375$ TB of memory. The cache of each CPU is made of 8×64 KB of L1 instructions, 16×16 KB of L1 data, 4×2 MB of L2 and 2×8MB of L3. The nodes are equipped with Mellanox ConnextX-3 QDR HCAs linked through 36-port Mellanox InfiniBand switches. All the nodes run the 2.6.32 version of the Linux kernel and MVAPICH2-1.7 over OFED 1.5.3.3. In all the tests, the linked list design, which is the default MVAPICH implementation, is used as the reference for speedup, memory overhead and cache performance results.

4.6.1 Notes on Measurements

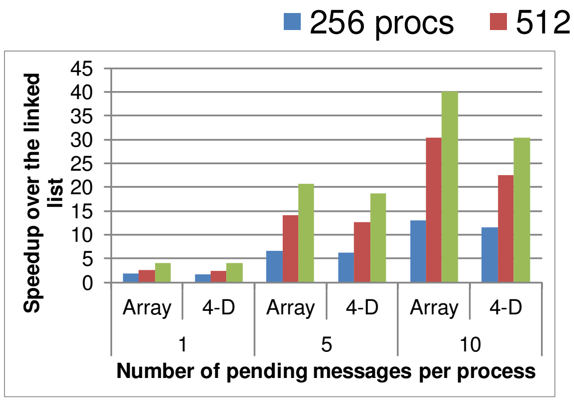
We measure the memory consumption of the actual queue operations inside each process for each queue architecture. The MVAPICH2 code implementation of each queue design has been instrumented for that purpose. Each allocation of objects such as RankHead, ContextIdLead, Cube, JumpPoint and MQI updates the average and maximum memory consumptions associated with message queue activities. For each message queue design, we use a separate MVAPICH build for memory tests; in order to avoid any instrumentation-induced impact on the latency results. Finally, we gather cache statistics with the Linux "perf stat" utility. The results are gathered for the whole execution of each process. The cache miss improvement for the array for instance is computed as $(cacheMissLinkedList - cacheMissArray) \times 100/cacheMissLinkedList$. The same formula is used for the 4D design. A 75% cache miss ratio improvement for the array for instance means that $cacheMissArray$ is only 25% of $cacheMissLinkedList$ for the same execution.

4.6.2 Microbenchmark Results

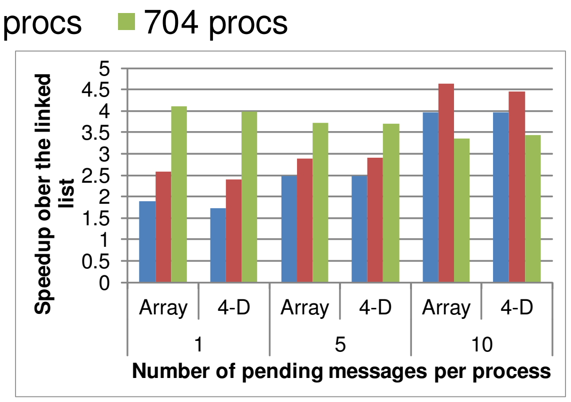
The microbenchmark program contains a single receiver and $n - 1$ senders in an n-process job setting. The goal is to build a certain length of PRQ or UMQ before the receiver starts searching through them. We use a special tag value to pass a synchronization token that enforces a ring ordering of send or receive message posting. To build a long PRQ, the token is first given to the receiver which posts all its receives before the token circulates. To build a long UMQ, the token is given to the receiver only after all the senders are done with it. All the microbenchmark metrics are gathered over the receiver process only; that is, the one that builds the long queues.

Single Communicator Tests

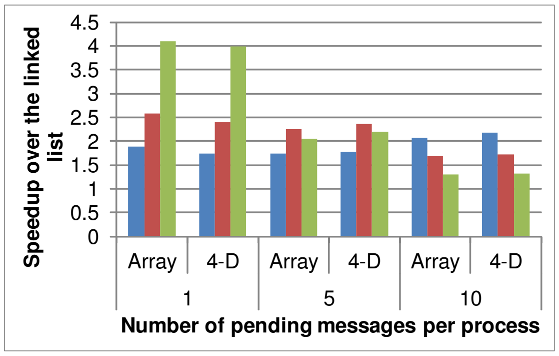
We first perform the tests with a single active communicator (i.e., one contextId) in the job. Figure 4.12 and Figure 4.13 show the queue operation time speedup yielded by the array-based and our 4D approaches for the PRQ and the UMQ respectively. When the queue items are consumed from the bottom of the queue (*reverse search*) as in the test described in



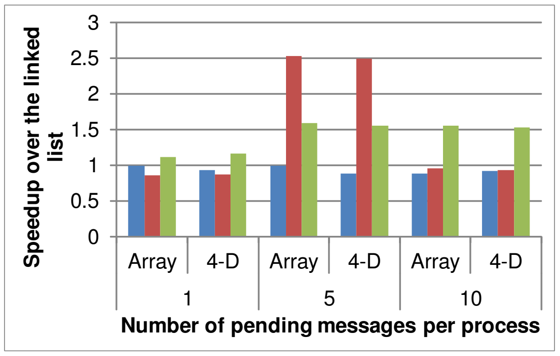
(a) Reverse search 0% MPI_ANY_SOURCE



(b) Reverse search 25% MPI_ANY_SOURCE



(c) Reverse search 50% MPI_ANY_SOURCE

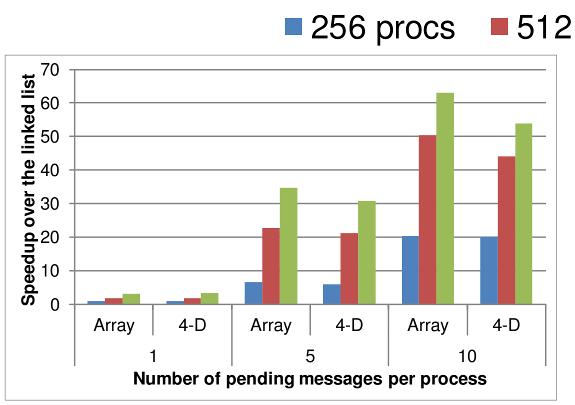


(d) Forward search

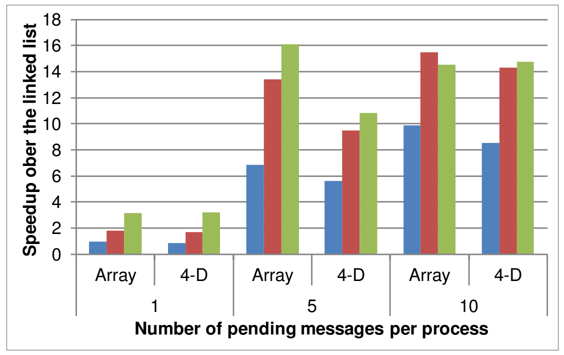
Figure 4.12: PRQ operation speedup over the linked list, with 1 communicator

[4], the speedups are always above 1. For 10 pending messages per sender, the PRQ search is accelerated by more than 40 times by the array-based design and more than 30 times by our 4D design (Figure 4.12(a)). The UMQ is accelerated even more by both the array and the 4D design (Figure 4.13(a)). The reverse search is the worst possible case for the linked list design because the amount of traversal is maximized. In contrast, the linked list leaves no room for performance improvement for *forward searches* where the item of interest is always at the top (Figure 4.12(d), Figure 4.13(d)). It can even be noticed that the speedups of the 4D and the array approaches are sometimes slightly below 1.0 for forward searches because their fixed multi-level traversal cost might not be compensated for.

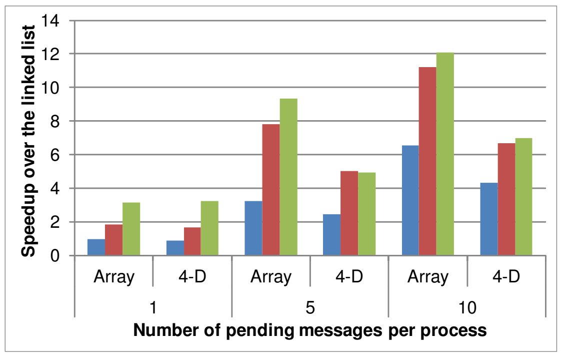
Those two forward and reverse search tests are the extremes. There are two kinds of in-between situations. The first one, which is not presented because it behaves like certain cases of the second one, is when a fraction of the matches occurs towards the bottom and



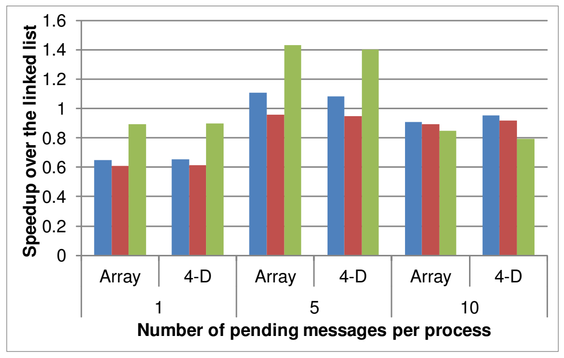
(a) Reverse search 0% MPI_ANY_SOURCE



(b) Reverse search 25% MPI_ANY_SOURCE



(c) Reverse search 50% MPI_ANY_SOURCE



(d) Forward search

Figure 4.13: UMQ operation speedup over the linked list, with 1 communicator

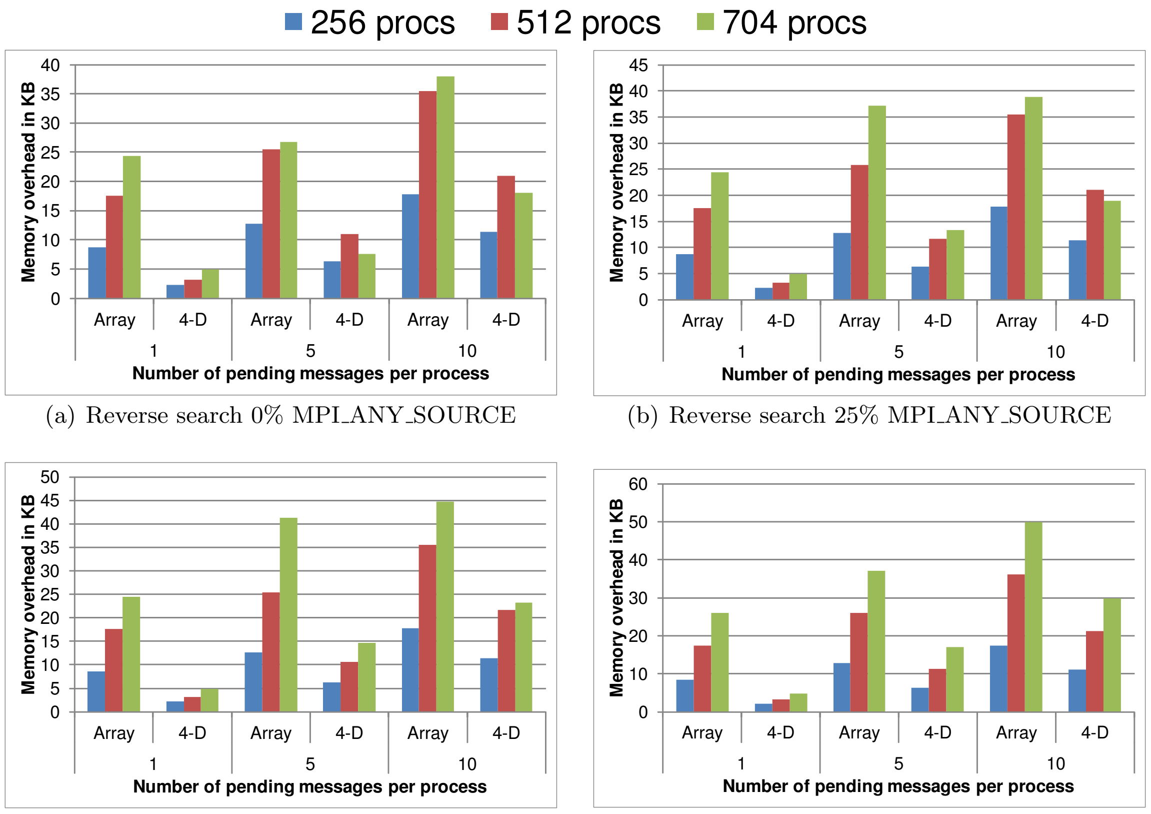
another fraction occurs towards the top. The second situation occurs when a fraction of the receives are `MPI_ANY_SOURCE` (Figure 4.12(b) and Figure 4.12(c) for PRQ; Figure 4.13(b) and Figure 4.13(c) for UMQ). `MPI_ANY_SOURCE` matches tend to produce three effects. First, they reproduce the forward search scenario in the PRQ as their matches occur at most after the number of pending tags per sender; that is, 1, 5 or 10 queue items in Figure 4.12, Figure 4.13. Second, because `MPI_ANY_SOURCE` sub-queues are strictly linear no matter the queue design, they increase the fraction of linear searches. Then in the UMQ, `MPI_ANY_SOURCE` receives tend to perform complete searches against all the existing ranks. While those combined effects quickly decrease the search improvements, the faster approaches remain better than the linked list unless the fraction of `MPI_ANY_SOURCE` reaches 100%; in which case the speedups disappear. Forward search results with fractions of `MPI_ANY_SOURCE` are omitted because they add little information to the set of tests already presented in Figure 4.12, Figure 4.13. All forward searches tend to behave like Figure 4.12(d) for the PRQ and Figure 4.13(d) for the UMQ.

Figure 4.14 and Figure 4.15 show the average memory overheard generated by each of the array and 4D approaches compared to the linked list. We omit the peak memory tests because they present the same trends as in Figure 4.14 and Figure 4.15. We can observe that the 4D approach is always the one that adds the smaller memory overhead to the linked list memory requirements. Even in the worst case, the 4D approach is approximately twice more memory efficient than the array design. We expect that ratio to keep improving with job sizes.

We gathered cache miss improvements ratios as well. However, for a single communicator, no specific trend is observed. The linked list behaves better or worse than the array and the 4D approaches in a random fashion. Based on the cache results that we show later for the multiple communicator tests, we can conclude that a clear trend builds up after a threshold that the single communicator tests has not reached in our microbenchmarks. In theory, every other thing kept constant, the cache behaviour is strongly impacted by the search depths. Consequently, even for a single communicator, we would expect a clear trend for job sizes larger than the one used in our tests.

Multiple Communicator Tests

We repeat the tests for 10 and 50 active communicators. The behaviour of the message queues in presence of `MPI_ANY_SOURCE` can be generalized from the single communicator tests. As a consequence, for conciseness sake, we do not repeat the tests with various percentages of `MPI_ANY_SOURCE`. Then, in the complete absence of `MPI_ANY_SOURCE`, both PRQ and UMQ behave very similarly. We therefore stick to only one of these queues, namely the PRQ. All the observations made for the PRQ with 10 and 50 communicators in absence of `MPI_ANY_SOURCE` can be generalized to the UMQ for the same number of communicators. All the tests are done for 5 pending messages per process. For each of the tests, we provide the results for forward and reverse traversal of the communicators. The forward traversal uses the same communicator traversal order for both senders and receivers. The reverse traversal does the opposite. These communicator traversal orders are different from the forward and reverse searches that happen inside each communicator. Figure 4.16 depicts the various matching sequences yielded by the



(a) Reverse search 0% MPI_ANY_SOURCE

(b) Reverse search 25% MPI_ANY_SOURCE

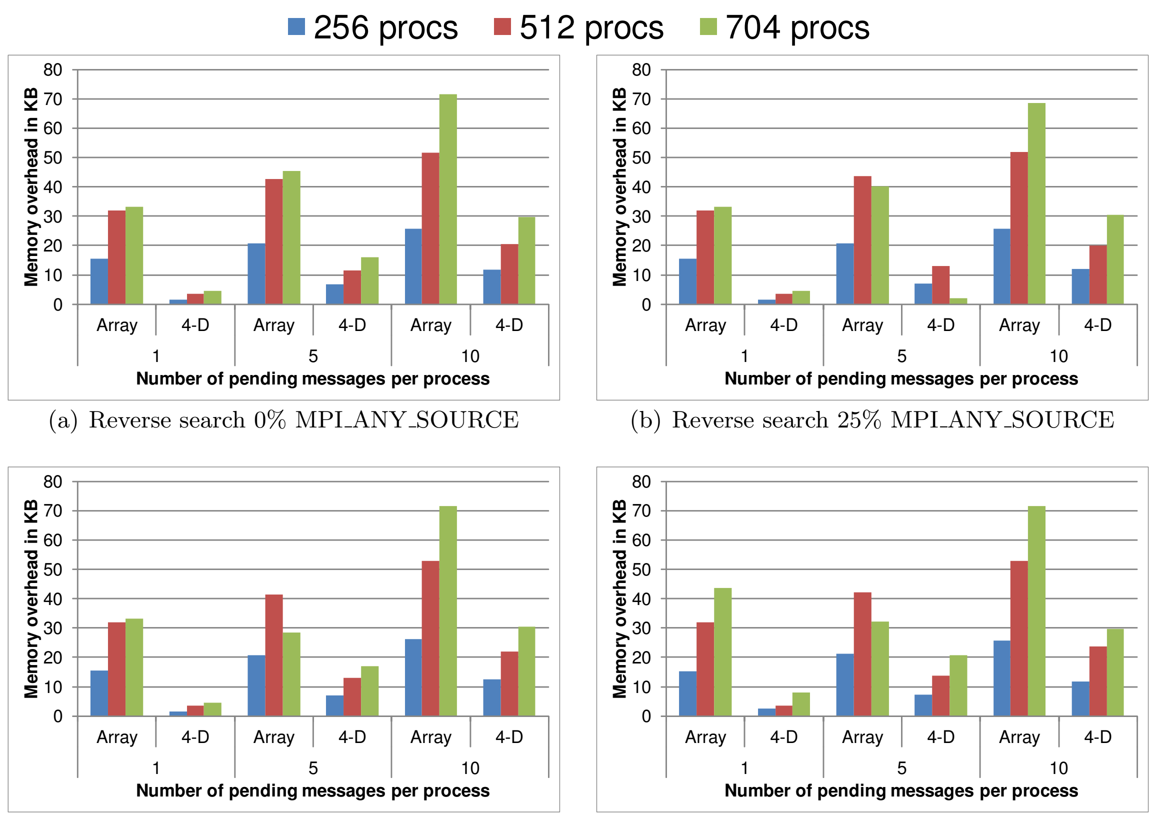
(c) Reverse search 50% MPI_ANY_SOURCE

(d) Forward search

Figure 4.14: PRQ average memory overhead compared to the linked list, with 1 communicator

combinations of those two distinct order parameters. As shown in Figure 4.16, the matches happen entirely for one rank before moving on to the next one.

We first notice that the speedup increases with the number of active communicators (Figure 4.17). The higher the number of communicators, the larger the speedups. The PRQ reverse search speedup with one communicator (Figure 4.12) varies between 1.7 and 40. With 10 communicators, it varies between 42 and 175.8 (Figure 4.17(a)). With 50 communicators, it varies between 100 and 556 (Figure 4.17(c)). The large speedups observed for multiple communicator tests can be explained as follows. The array and 4D approaches always skip all the queue items in any communicator that is not of interest for the search in progress. In comparison, the linked list approach always scan the queue linearly and go over all the queue items of the irrelevant communicators if the communicator of interest happens to occur later in the queue.



(a) Reverse search 0% MPI_ANY_SOURCE (b) Reverse search 25% MPI_ANY_SOURCE

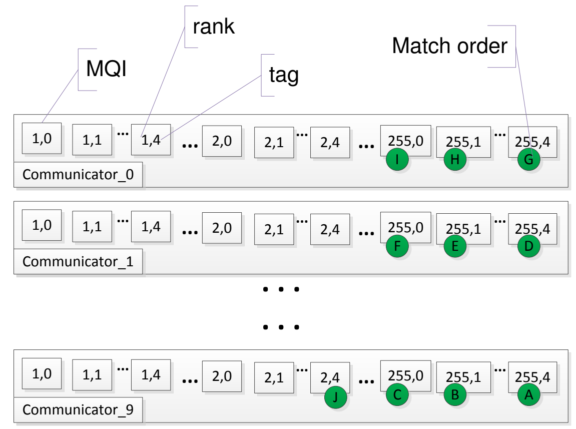
(c) Reverse search 50% MPI_ANY_SOURCE (d) Forward search

Figure 4.15: UMQ average memory overhead compared to the linked list, with 1 communicator

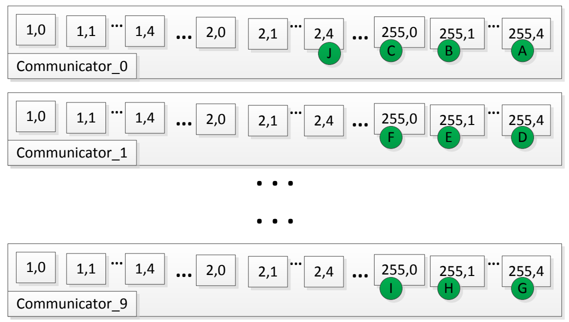
As shown in Figure 4.16, the linked list always begin with the queue item (1, 0) of communicator_0, go all the way to the end of that communicator and then jumps on queue item (1, 0) of communicator_1 and then progresses the same way until the match occurs. The magnitude of the speedups observed in these tests is an embodiment of the observation number 4 at the end of Section 4.4.1. The same information is conveyed by Table 4.3 in Section 4.4.1.

The average memory overhead tests (Figure 4.18) show the expected trend of the array being more memory-greedy than the 4D approach. Once again, we omit the peak memory overhead tests because they present the exact same trend as the ones shown in Figure 4.18.

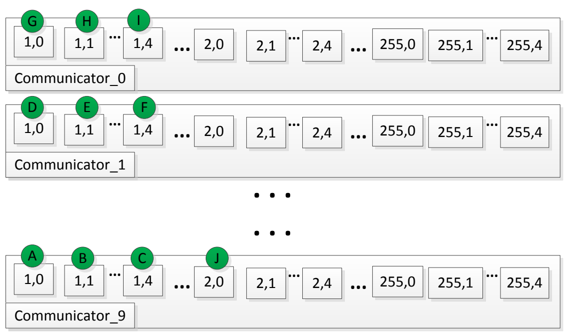
We notice substantial cache miss ratio improvements (Figure 4.19, Figure 4.20, Figure 4.21). As expected, the L1 data cache miss improvement is higher with 50 communicators (Figure 4.19(c), Figure 4.19(d)) than with 10 communicators (Figure 4.19(a), Figure 4.19(b)). For the L1 instruction cache miss improvements, the tests with 10 communicators (Figure 4.20(a),



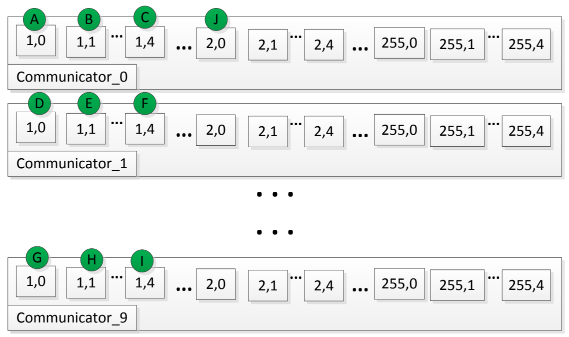
(a) Reverse search AND reverse communicator traversal



(b) Reverse search AND forward communicator traversal



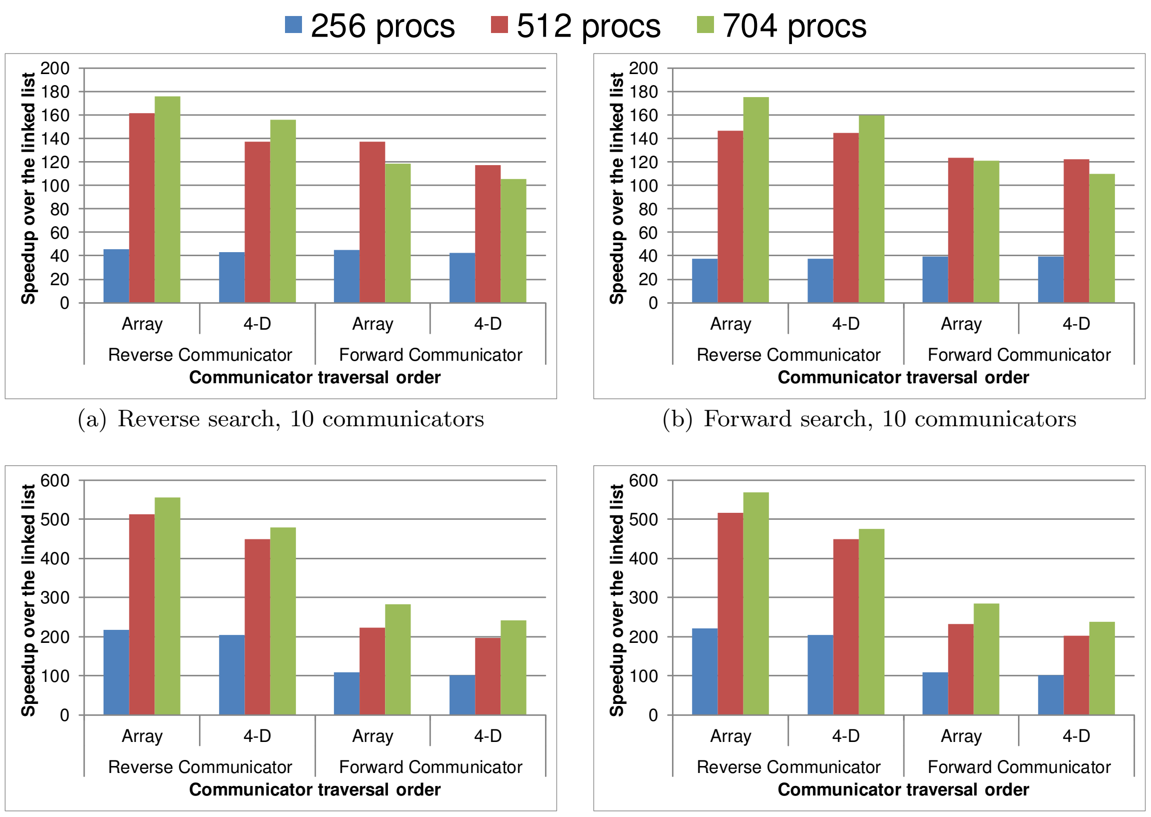
(c) Forward search AND reverse communicator traversal



(d) Forward search AND forward communicator traversal

Figure 4.16: Different match orders for multi-communicator tests: example with 10 communicators of 256 distinct ranks and 5 pending messages per process

Figure 4.20(b)) show a slight absence of consistent trend even though the general observation is an improvement. With 50 communicators (Figure 4.20(c), Figure 4.20(d)), the trend for L1 instruction cache miss improvement is clear and the improvement ratios are higher as well. Finally the last level cache ratios (Figure 4.21) follow the same trend of increase from 10 to 50 communicators. The tests in Figure 4.21(c) and Figure 4.21(d) correspond to high rates of back-and-forth between the L3 cache and physical memory for the linked list-based queue; a very detrimental situation that both the array and the 4D approaches fix.



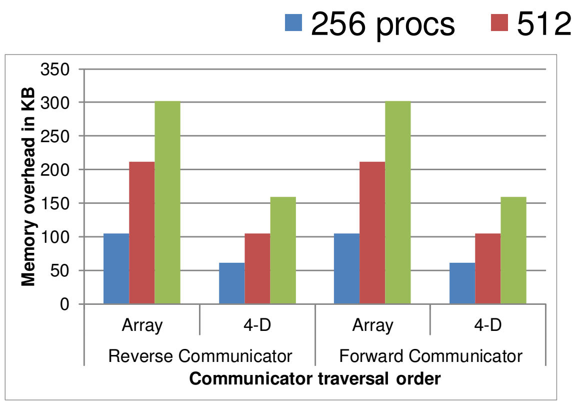
(a) Reverse search, 10 communicators

(b) Forward search, 10 communicators

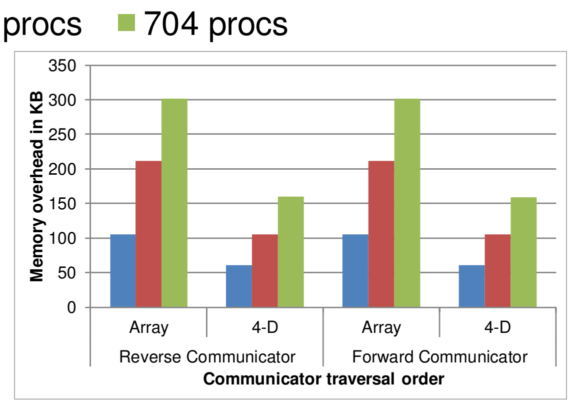
(c) Reverse search, 50 communicators

(d) Forward search, 50 communicators

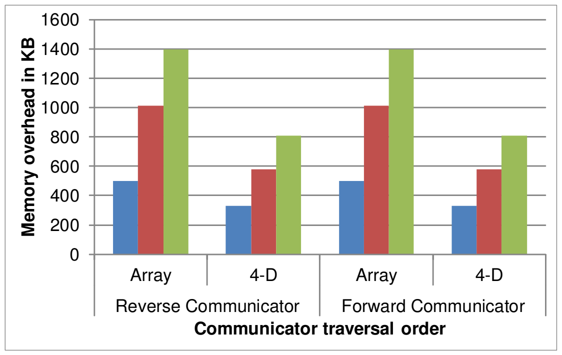
Figure 4.17: PRQ operation speedup over the linked list, with multiple communicators



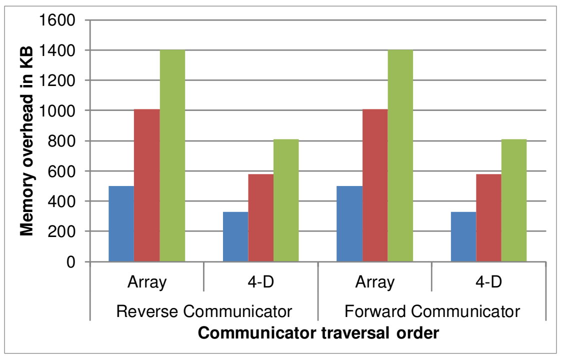
(a) Reverse search, 10 communicators



(b) Forward search, 10 communicators

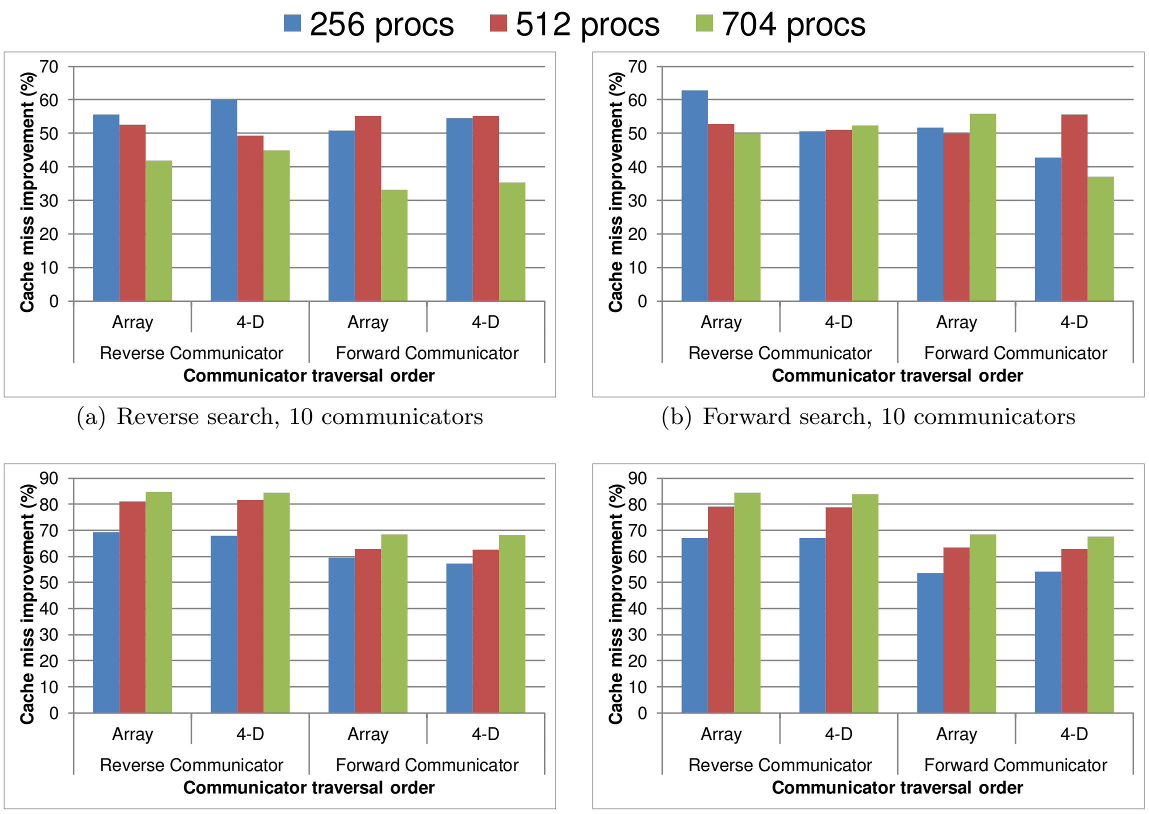


(c) Reverse search, 50 communicators



(d) Forward search, 50 communicators

Figure 4.18: PRQ average memory overhead compared to the linked list, with multiple communicators



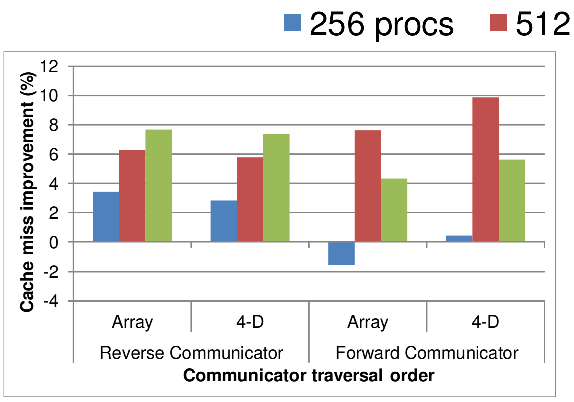
(a) Reverse search, 10 communicators

(b) Forward search, 10 communicators

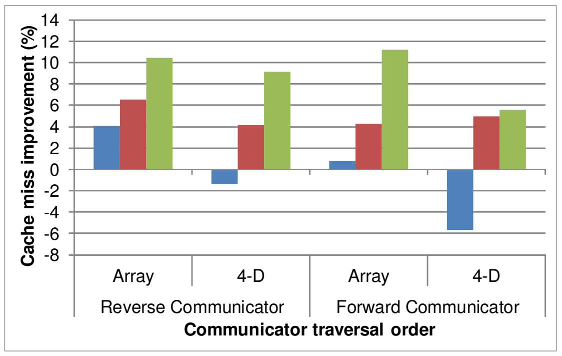
(c) Reverse search, 50 communicators

(d) Forward search, 50 communicators

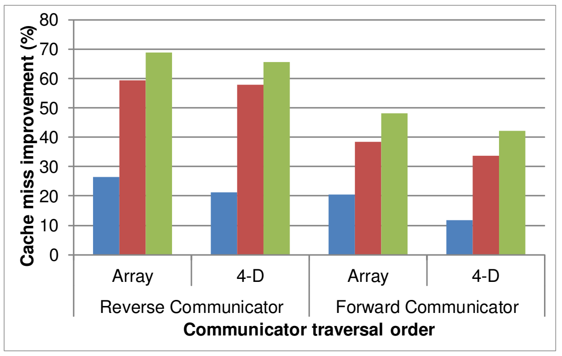
Figure 4.19: PRQ L1 data cache improvement over the linked list, with multiple communicators



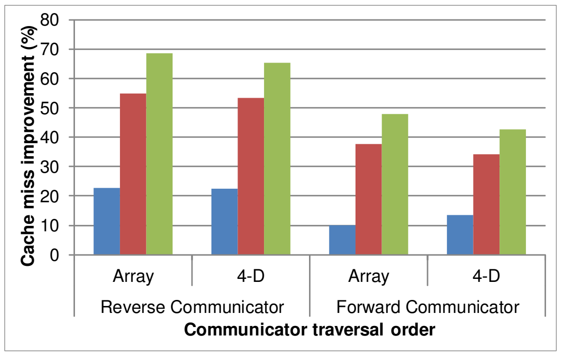
(a) Reverse search, 10 communicators



(b) Forward search, 10 communicators

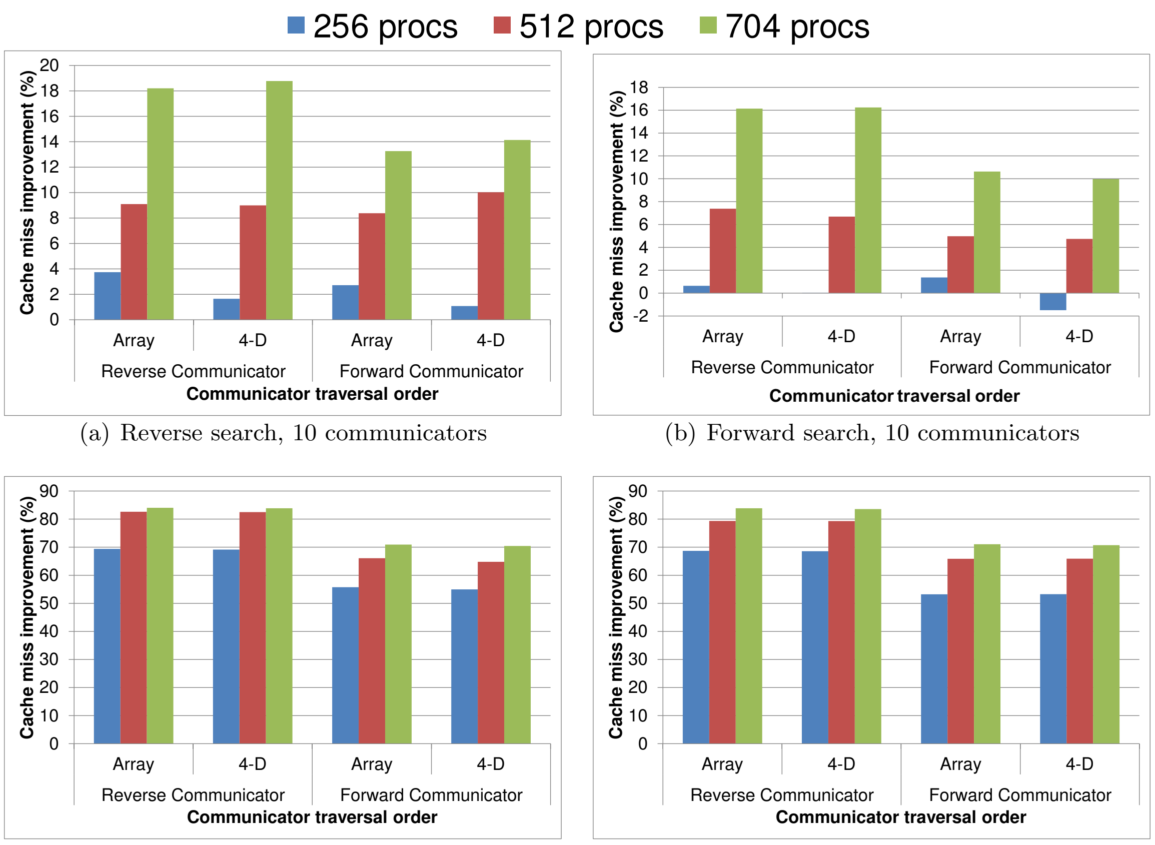


(c) Reverse search, 50 communicators



(d) Forward search, 50 communicators

Figure 4.20: PRQ L1 instruction cache improvement over the linked list, with multiple communicators



(a) Reverse search, 10 communicators

(b) Forward search, 10 communicators

(c) Reverse search, 50 communicators

(d) Forward search, 50 communicators

Figure 4.21: PRQ last level (L3) cache improvement over the linked list, with multiple communicators

4.6.3 Application Results

The tested applications are Radix [92] and Nbody [2, 92]. Radix is a sorting algorithm. The application kernel used in the tests sort 64-bit integers. Nbody is a particle interaction simulation. Radix and Nbody are interesting for the queue issue because they exhibit the two cases of superlinear and linear queue growth respectively. We emphasize that we did test our implementations for correctness and ordering constraints assessment with applications that use `MPI_ANY_SOURCE`. The tested applications additionally have very shallow search depths and present no room for search optimization. We experimented with NAS LU [74] and AMG2006 [36] as they build reasonably long queues. NAS LU and AMG2006 are linear algebra applications. Unfortunately, AMG2006 over 512 processes has average search depths of 4.29 and 3.56 for UMQ and PRQ, respectively. These metrics are 4.04 and 0.67 for NAS LU class E [74] over 512 processes. In general and regardless of the presence of `MPI_ANY_SOURCE`, applications with shallow search depths like AMG2006 nd NAS LU offer no room for improvement for the array or 4D message queues compared to the linked list. Very small queue depths correspond to the forward search microbenchmark scenarios and can even lead to a slight degradation when the linked list message queue is not used.

For Radix, the radix parameter is 16. The application is run respectively to sort 2^{28}, 2^{29} and 2^{30} 64-bit integers, as presented in the first column of Table 4.8 and Table 4.9. All three executions are done over 512 processes because our experiments show that the job size is less of an impact than the data size for Radix. Nbody is run over 256, 512, and 704 processes, respectively. The first column of Table 4.8, Table 4.9 show the job sizes appended to the names of Nbody. For the PRQ and the UMQ, we provide respectively in Table 4.8 and Table 4.9 some queue behaviour metrics that are used in the analysis of the performance results. The values in the tables are the averages computed over three iterations; this explains why certain maximum lengths are not integers. The Max Queue Length (MQL) represents the maximum queue length reached in the application. The UMQ MQL of Radix reaches several times its job size. As for Nbody, both its PRQ and UMQ grow almost linearly with the job size. Its PRQ MQL is actually exactly $JobSize - 1$ (Table 4.8). The Max Average Search Length (MASL) column shows the largest of the average search lengths returned by each process. The Average

Table 4.8: PRQ behaviour data for Radix and Nbody

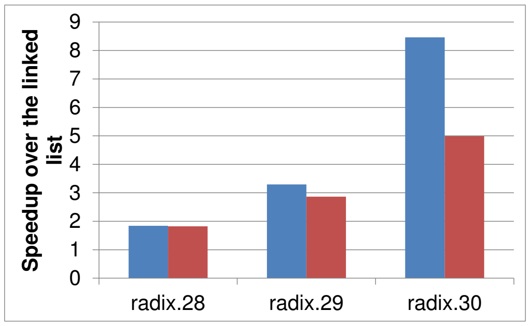
Application and parameters	Max Queue Length (MQL)	Average Queue Length (AQL)	Max Average Search Length (MASL)	Average Search Length (ASL)
Radix.n28	4	4	0.998	0.817
Radix.n29	4	4	0.998	0.823
Radix.n30	4	4	0.979	0.832
Nbody.256	255	254.953	0.981	0.926
Nbody.512	511	508.775	0.984	0.867
Nbody.704	703	682.646	0.982	0.796

Table 4.9: UMQ behaviour data for Radix and Nbody

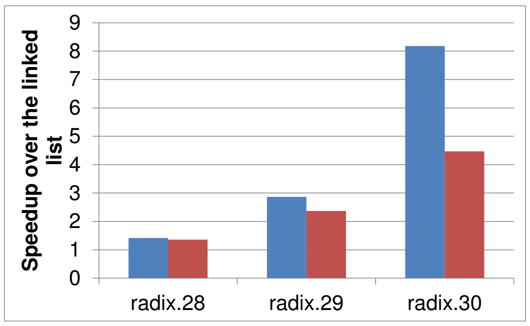
Application and parameters	Max Queue Length (MQL)	Average Queue Length (AQL)	Max Average Search Length (MASL)	Average Search Length (ASL)
Radix.n28	3061.3	484.905	1510.058	46.547
Radix.n29	6110	492.449	3418.630	54.468
Radix.n30	13228	533.837	7208.657	74.740
Nbody.256	249.333	86.863	27.228	6.041
Nbody.512	498.000	370.342	32.363	17.718
Nbody.704	693.667	551.003	50.863	21.876

Search Length (ASL) shows the average search length over the whole job. The bigger these two metrics, the closer the search behaviour gets to the single communicator reverse search scenario presented in the microbenchmarks (Section 4.6.2); and the smaller they are, the closer it gets to the forward search scenario. Table 4.9 shows that the UMQ MASL of Radix reaches several thousands, making this application a good example of situation where fast queue traversal can make a noticeable difference.

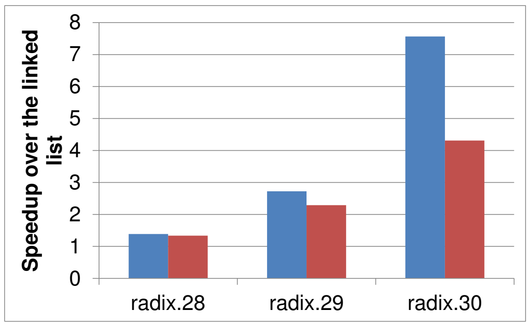
Application results are gathered per process; then averaged over all the processes of the job. For instance, peak memory represents the average of the peak memory outputted by each individual process; and average memory is averaged over the average memory outputted by each process. Figure 4.22 and Figure 4.23 present the performance results for Radix and Nbody respectively. For Radix, the queue operation speedup reaches 8.46 for the array-based approach and 4.99 for the 4D approach (Figure 4.22(a)). More importantly, the queue operation speedups are conveyed almost entirely to the communication and execution time improvements (Figure 4.22(b), Figure 4.22(c)). For 2^{30} integers in particular, the array-based and the 4D approaches tremendously improve the overall execution time of Radix.



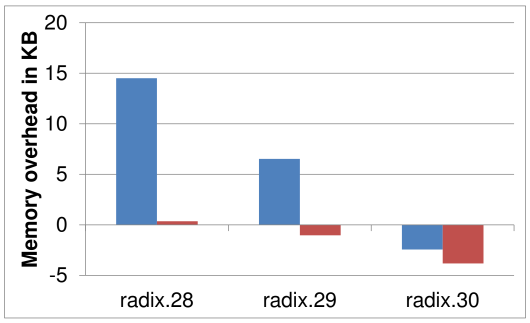
(a) Queue operation time speedup



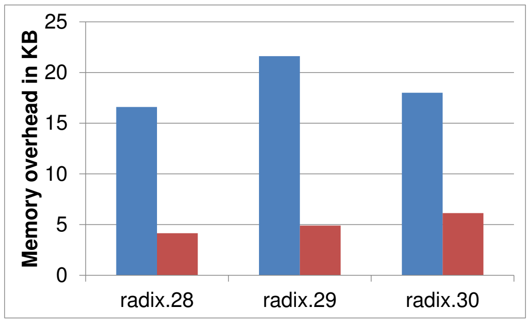
(b) Communication time speedup



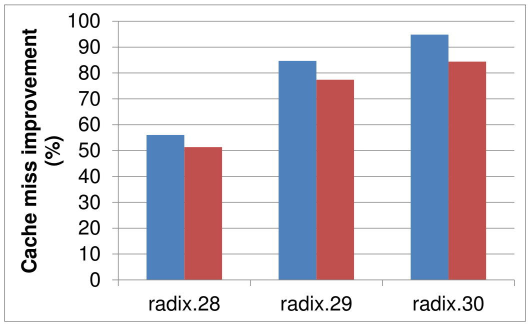
(c) Execution time speedup



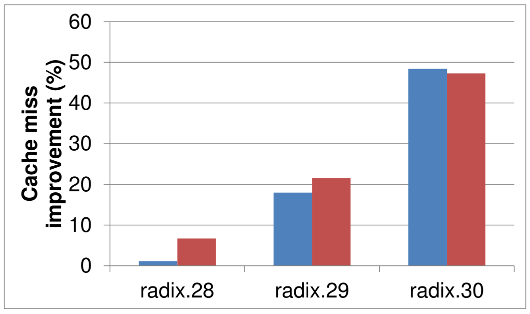
(d) Average memory overhead



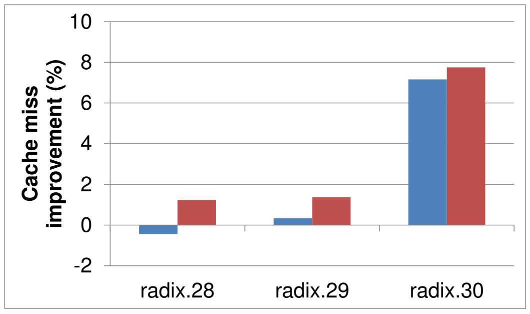
(e) Peak memory overhead



(f) L1 data cache miss improvement

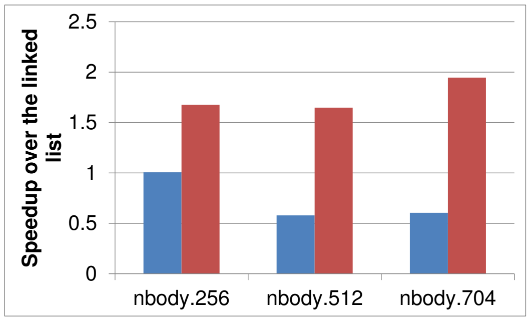


(g) L1 instruction cache miss improvement

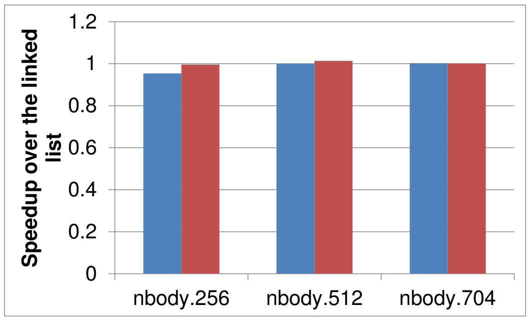


(h) Last-level cache miss improvement

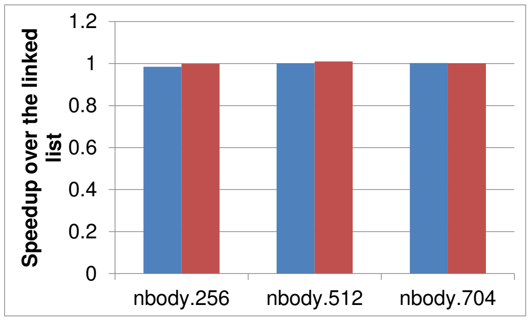
Figure 4.22: Radix speedups, memory overhead and cache statistics for the array-based and 4D designs over the linked-list approach



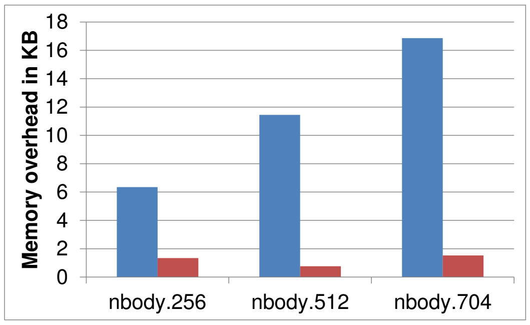
(a) Queue operation time speedup



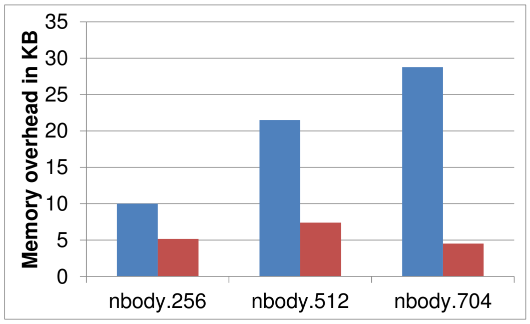
(b) Communication time speedup



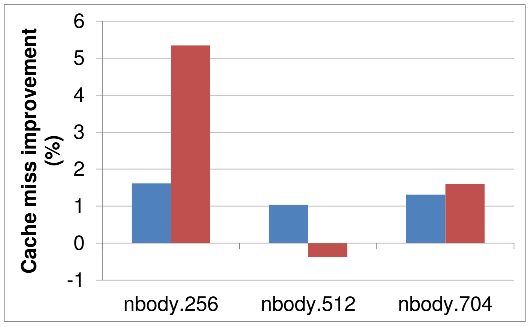
(c) Execution time speedup



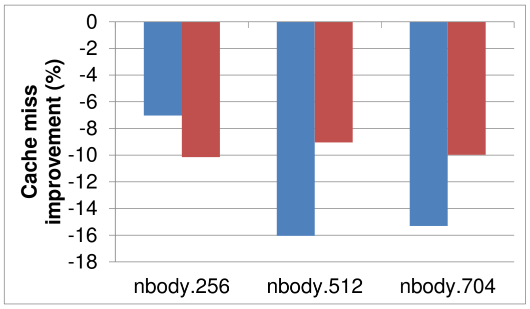
(d) Average memory overhead



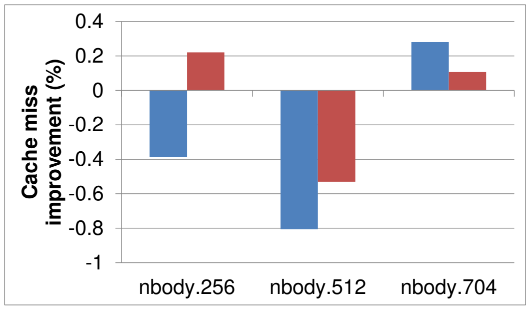
(e) Peak memory overhead



(f) L1 data cache miss improvement



(g) L1 instruction cache miss improvement



(h) Last-level cache miss improvement

Figure 4.23: Nbody speedups, memory overhead and cache statistics for the array-based and 4D designs over the linked-list approach

These large speedups in communication and execution times are due to the process of rank zero being a bottleneck and propagating the resulting latencies when the linked list queue is used. The MASL of Radix comes from its rank 0 and completely overshadows its ASL to be the key factor in its performance.

For Nbody (Figure 4.23(a)), the 4D approach accelerates the queue processing time by close to 2 times. The array-based approach degrades the processing time instead. The array approach actually incurs an overhead when the queues become completely empty. The deallocation that ensues can be costly when it is frequent. These effects might not be compensated for when the search lengths are not large enough. The 4D approach is shielded against that deallocation effect because it uses very small objects that are created and freed from various pools. Figure 4.22(b) and Figure 4.22(c) show that the 4D approach improvement and the array-based design degradation do not noticeably impact the communication and execution times of Nbody. Nbody is very balanced as shown by the very small differences between the max metrics (MQL, MASL) and the average metrics (AQL, ASL) in Table 4.8, Table 4.9.

As for the memory overheads, there is a clear trend that the array-based approach is always more memory-intensive than the 4D design (Figure 4.22(d), Figure 4.22(e) and Figure 4.23(d), Figure 4.23(e)). The memory overheads, both average and peak values, are very consistent with the expectations. For Nbody (Figure 4.23(d)) there is a clear increase of memory for the array when the job size increases. In comparison, the 4D approach does not show that linear memory overhead growth. With Radix (Figure 4.22(d)), the 4D approach shows a decreasing average memory overhead when the number of sorted integers grows. There is a negative memory overhead effect when there is less queue build-up due to the speed of processing. The memory overhead of a faster approach can end up being more than compensated for because the average number of items in the queues is kept low. The negative memory overhead effect is fairly present with Radix.n30 (Figure 4.22(d)) when executed with the 4D approach. The array approach can yield the negative memory overhead to a lesser extent than the 4D approach because of its fixed allocation scheme that depends on the communicator size. We must emphasize a key difference between the applications and the microbenchmarks for the memory behaviours. With the microbenchmarks, the queues of the linked list, the array and the 4D container all have the same number of items at each step of the execution. Actually, the microbenchmarks do simple back-to-back communications for the purpose of message queue processing. The applications on the other hand mix computations and communications in a totally uncontrolled fashion as far as message queue operations are concerned. In particular, the three kinds of message queue do not necessarily have the same number of queue items at the same steps of the application execution. The negative memory overhead and the difference of trends between the peak and average memory overheads are a consequence of the uncontrolled environment shown by the applications. The observation being made and explained here is that the memory behaviour of the 4D approach is more interesting in non-controlled environments; the very environment that characterize applications.

Finally, Radix generally improves a lot on the L1 data cache misses. The instruction cache miss rate improvement is substantial as well; especially when sorting 2^{30} values. As expected, the last-level cache gets noticeably improved for Radix.30. The comparative cache miss variations are not pronounced for Nbody. There is generally a slight improvement for L1 data cache and a generally slight degradation of less than 0.8% in the worst case for the last-level cache. These small improvement and degradations are due to the relatively shallow search depth shown by Nbody. The L1 instruction cache for Nbody, however, shows serious degradations for both the array and the 4D containers. Nbody is known to be very computation-intensive [2]. As a result, its instruction cache is usually filled with non-message queue-related code. The linked list-based message queue instructions are simpler and less diversified than the array and 4D-based queue operation instructions. Consequently, when the queue operation instructions must get room in the L1 instruction cache or leave it for computation instructions to occupy the space, less eviction happens with the linked list than the other two message queue types.

4.6.4 Message Queue Memory Behaviours at Extreme Scales

While the speed differences between the 4D and the linked list-based queues are obvious even at 512 and 704 processes, the memory values obtained convince only by their trends (see Table 4.10). In fact, the cluster that we used for these tests has 2 GB of memory per CPU core;

Table 4.10: Average memory overhead measurements of the applications (in KB)

App	Linked list	Array	4D	Overhead Array	Overhead 4D
Radix.28	97.16	111.66	97.51	14.50	0.36
Radix.29	108.47	114.99	107.44	6.53	-1.030
Radix.30	141.07	138.61	137.01	-2.44	-3.81
Nbody.256	79.71	86.06	81.05	6.35	1.34
Nbody.512	82.11	93.56	82.88	11.44	0.76
Nbody.704	107.05	123.91	108.57	16.86	1.53

and the array average memory overhead for Nbody at 704 processes represent only 0.0008% of the memory available for the process.

However, the scales targeted by this work go far beyond 704 CPU cores. It helps to examine the percentage of process memory consumed by message queues on two petascale supercomputers, namely the Titan-Cray XK7 and the Blue Gene Sequoia [106]. Radix was run at the fixed size of 512 processes and its results are difficult to extrapolate with larger system sizes. Nbody was run over increasing job sizes and is therefore a good candidate for extrapolation. Each of the 1,572,864 CPU cores of Blue Gene Sequoia has no more than 1 GB of RAM available while each of the 560,640 CPU cores of the Titan-Cray XK7 has less than 1.27 GB. Let us also consider a large hypothetical HPC application A_H which reuses about 2 libraries built out of MPI. That application would have a total of 3 job-size communicators [41], leading to a total of 6 job-size contextIds in an MPI library such as MVAPICH or MPICH where the same communicator has two contextIds [1]; one for collectives and another for non-collective communications. We are interested in seeing the impact of running Nbody or the hypothetical job A_H on the Titan-Cray XK7 and the Blue Gene Sequoia.

We show in Table 4.11 the absolute message queue-related memory consumption and the percentage of the physical memory represented by that consumption for each process. Our 704-CPU core test system is also represented (”Current”) in Table 4.11 for comparison purposes. Anticipating the same linear growth observed for the average memory overhead of Nbody for the array (Table 4.10), we extrapolate that same amount of overhead to Titan and Sequoia. For the 4D, the extrapolation is a more difficult task because the memory overhead growth is not linear (both in theory and experimentally). For instance the 4D queue consumes less memory for 1,048,576+1 than for 1,048,576 because of the change in *dimensionSpan* (see Table 4.5 for more such examples). Nevertheless, we show for the application A_H what would happen. The results shown for A_H are computed from the data in Table 4.4, along with Equation 4.19 and Equation 4.20. For Nbody, the fixed memory overhead of the array would represent more than 3% of the available RAM for each process on Sequoia. That overhead exists even when the message queues host only 1 MQI. However, the process must fit, in that 1 GB, its binary and all the other MPI internal data; on top of hosting its useful application data; without hopefully swapping. It helps to mention that MPI itself already needs a substantial amount of system buffers to function. Those buffers are crucial for the unavoidable Eager and control messages. The requirement of these buffers is even higher for RDMA-enabled network technologies such as InfiniBand and iWARP. For those particular network technologies, the lesser system buffer MPI can access, the stricter the flow control mechanism it must use to recycle them. The effects of this flow control mechanism are wait and delay propagation. It also helps to realize that the fear of memory consumption at large scale in MPI is not necessarily viewed from the point of view of a single MPI aspect such as message queues; it is the aggregated effect of many aspects that is worrisome. This approach of reducing memory footprint from different perspectives is already being fulfilled in the community; and justifies ideas like replacing comprehensive enumeration of process ranks by ranges to save on memory consumed by, not array-based message queue RankHeads, but mere 8-byte integers [3]. All those considerations show that the 28% fixed memory overhead of the array approach for A_H , compared to less than 1% for the 4D, would be unsustainable for Sequoia, let alone for future supercomputers. We emphasize that the description of A_H is completely realistic. Actually, we even omitted in its definition the many smaller communicators that an informed MPI programmer might create in such large jobs to perform localized communications in smaller subgroups of processes. Those smaller communicators would worsen even further than described in Table 4.11 the impact of array-based message queues on memory per CPU core at large scales.

Table 4.11: Memory consumption overhead extrapolation on Sequoia and Titan

	Nbody		Application A_H	
	Array overhead in GB (% of RAM per process)	4D overhead in GB (% of RAM per process)	Array overhead in GB (% of RAM per process)	4D overhead in GB (% of RAM per process)
Current	1.608E-05 (8.04E-4%)	1.456E-6 (7.278E-5%)	1.26E-4 (6.30E-3%)	2.5E-5 (1.20E-3%)
Titan	0.0128 (1.01%)	-	0.100251 (**7.89%**)	4.727E-3 (**0.37%**)
Sequoia	0.036 (3.59%)	-	0.281250 (**28.10%**)	6.610E-3 (**0.66%**)

4.7 Summary

With more and more jobs expected to run on millions of CPU cores in the petascale and exascale computing eras, MPI distributions must fix many little details that sink performance and increase memory consumption at scales. The MPI message queue mechanism is one of those details. The message queues are on the critical path of MPI communications; and a good implementation cannot afford to leave them unscalable.

From a speed of operation point of view only, it is possible to adopt an array-based design for which the operations which are usually expensive happen in $O(1)$. Unfortunately, this comes at the expense of increased memory consumption at large scales. As physical memory is a shrinking resource per CPU core, the array-based approach is a solution that bears another scalability issue at large scales because it wastes a lot of memory for large communicators.

We propose in this chapter a novel message queue mechanism that is scalable with respect to both speed and memory consumption. After separating the queue items by communicator identifiers, our design decomposes the rank into four slices and uses the three most significant slices to build a 3-dimensional jump coordinate that is used to substantially accelerate searches in situations of large queues. A fourth dimension made of the least significant slice is exploited to build small-length PRQs and UMQs. Our 4-dimensional design not only limits considerably linear traversals but it optimizes unfruitful searches by detecting them early in the 3-dimensional jump process. We investigated well-known data structures such as binary search trees but they proved to be slower and substantially memory-greedier than the proposed 4-dimensional message queue mechanism. The new message queue proposed in this chapter is network technology-agnostic; it can therefore be ported to any MPI distribution that resorts to message queues in software.

We mentioned in Section 4.2.1 that RMA is the only MPI communication model that comes close to avoiding message queue buildup. In fact, since one-sided communications bear no concept of reception, they are in theory shielded against the message queue issue. Furthermore, unlike the two-sided model where two progress engines must be involved in fulfilling the communication, one-sided communications alleviate or avoid coupling between communicating peers. Such a positive characteristic is the very essence of its one-sided nature. In general, one-sided communications represent a paradigm whose positive advantages tend to nullify the issues discussed in both Chapter 3 and Chapter 4. However, the large amount of MPI-based HPC applications is based on two-sided and collective communications. The MPI one-sided communication model is not being used in the HPC community; and one reason for that lack of adoption lies in its inherent synchronization. In the next chapter, we show how one-sided communications, in spite of being well-suited for next generation supercomputing, fails to keep its promises in MPI. Then, we put forth a proposal to concretize the compelling one-sided communications traits that MPI-RMA lacks.

Chapter 5

Nonblocking MPI One-sided Communication Synchronizations

MPI is ubiquitous in the HPC community. However, in more than 15 years of existence, the one-sided communication model of MPI, also called MPI RMA, has not received the expected adoption. This situation is unfortunate because the one-sided communication model has an important semantic advantage over its two-sided counterpart. In fact, MPI RMA does not require a receive-side message matching; and as such, one-sided communications are less dependent on the availability of the destination process than two-sided message passing. This characteristic allows certain uses of MPI RMA that point-to-point and collective communications cannot facilitate [97, 120]. By allowing a single peer, the initiator, to make the communication, one-sided is the only MPI communication model suitable for unstructured dynamic communication patterns. MPI one-sided is therefore not just a more or less more performing alternative to two-sided communications; it brings its own additional features and flexibility to allow certain classes of computations to happen over MPI. Furthermore, the rising popularity of PGAS languages [19] and their suitability for a certain class of processing (e.g., Global Arrays [75]) tends to further support the reasonable assumption that the weak adoption of MPI RMA is not representative of its current and future importance.

MPI one-sided communications must happen inside critical section-like regions called epochs. An epoch is started by one of a set of RMA synchronization calls and ended by a matching synchronization call. MPI-2.0 RMA was criticized for its synchronization burden [10] and various constraints which make it unusable in certain situations. In that regard, MPI-3.0 represents an undeniable improvement as it first alleviates quite a few of the aforementioned constraints; and second, it introduces many new features which provides mechanisms to avoid frequent synchronization invocations. For instance, the introduction of the ***flush*** family of completion routines and the request-based one-sided communications in MPI-3.0 RMA voids the need for the communication initiator to always make a synchronization call to detect RMA completion at application level. However, these improvements are still not enough to make MPI RMA an equally compelling option compared to two-sided communications. For instance, when it must communicate with disjoint sets of targets, the communication initiator cannot avoid resorting to multiple distinct epochs; meaning that the new ***flush*** family of routines or the request-based communication functions cannot help. Serialization thus ensues because epoch-ending routines are blocking. The blocking nature of MPI one-sided communication synchronizations is actually the cause of six kinds of latency issues first documented in [55] as the MPI-2 RMA inefficiency patterns. These latency issues are the consequence of late peers propagating latencies to remote processes blocking on RMA synchronizations. Four of the inefficiency patterns are issues for which the specification of MPI-2.0 RMA offers no workaround. All four inefficiency patterns are still left unaddressed by the new MPI-3.0 specification.

In this chapter we effectively address the synchronization burden of MPI-3.0 RMA through a proposal which makes the one-sided communications nonblocking from start to finish, if desired [122]. Because the entire MPI-RMA epoch can be nonblocking, MPI processes can issue them and move on immediately. Asynchonicity is increased and conditions are created for enhanced communication/computation and communication/communication overlappings; but more importantly, the effects of late peer arrival can now be mitigated to a much higher extent by the application. The new proposal solves all four inefficiency patterns, plus a fifth one introduced and documented in the dissertation. Additionally, it allows various kinds of aggressive communication scheduling meant to reduce the overall completion time of multiple epochs initiated from the same process. As long as the invocation order of equivalent synchronization calls is maintained, a job written with the proposed nonblocking synchronizations could see the same write ordering in every process as if it was written with the current MPI-3.0 blocking synchronizations. To the best of our knowledge, this work represents the first attempt to remove all wait phases from the lifetime of MPI RMA epochs.

5.1 Related Work

The MPI specification gives a lot of leeway to implementers about when to force the waits in an epoch. In particular, an access epoch opening synchronization call does not have to block if the corresponding exposure epoch is not opened yet. This freedom is used in [7, 29, 104] to mitigate the apparent effects of RMA synchronization by deferring the actual internal execution of both synchronization and communication to the epoch-closing routine execution. This approach is termed *lazy*. Some early work on MPI one-sided communications did elect to block the origin at epoch-opening until the target is exposed or ready to be accessed [108]. While this approach imposes a wait on the origin, it avoids any buffering or queuing meant to defer the RMA communications. In [90] a design of the fence epoch is proposed where communication/computation overlapping occurs inside the epoch. However, since the proposal makes every single fence call blocking, the overlapping comes at the price of a potentially substantial idleness at both opening and closing of each epoch.

Purely intra-node RMA issues have been addressed in [58, 63]. An approach to the hybrid design of MPI one-sided communication on multicore systems over InfiniBand is presented in [89]. The work described a way of migrating passive target locks between network atomic operations and CPU-based atomic operations. The use of RDMA for one-sided communications was presented in [48, 61, 91]. Designs of the computational aspects of `MPI_ACCUMULATE` were proposed in [47, 76]. In [116], a strategy is proposed to adaptively switch between lazy and eager modes for RMA communications to achieve overlapping. An MPI-3.0 RMA library implementation for Cray Gemini and Aries systems is described in [27]. However, to the best of our knowledge, none of the previous work altered the epoch-closing routines of MPI RMA to render the one-sided communications lifetime nonblocking from start to finish. Thus, the work presented in this chapter pioneers entirely nonblocking MPI one-sided synchronizations proposals and designs.

5.2 The Burden of Blocking RMA Synchronization

5.2.1 The Inefficiency Patterns

The MPI one-sided inefficiency patterns [38, 55] are situations that force some unproductive wait or idleness on a peer involved implicitly or explicitly in RMA communications. The inefficiency patterns are a consequence of the blocking nature of RMA synchronization routines. They are listed as follows:

- *Late Post*: It is a GATS-related inefficiency where `MPI_WIN_COMPLETE` or `MPI_WIN_START` must block because the target is yet to issue `MPI_WIN_POST`. As a reminder, in a GATS epoch, `MPI_WIN_START` and `MPI_WIN_COMPLETE` respectively open and close the origin-side epoch. `MPI_WIN_POST` is invoked by the target to create an exposure epoch.

- *Early Transfer*: It occurs when an RMA communication call blocks because the target epoch is not yet exposed.

- *Early Wait*: It occurs when `MPI_WIN_WAIT` is called while the RMA transfers are not completed yet. As a reminder, `MPI_WIN_WAIT` does not exit until all the RMA transfers directed towards the calling process complete.

- *Late Complete*: It is the delay between the last RMA transfer and the actual invocation of `MPI_WIN_COMPLETE`. That delay propagates to the target as an unproductive wait. The target-side `MPI_WIN_WAIT` must block as long as `MPI_WIN_COMPLETE` is not invoked by the origin; and when that invocation is delayed for reasons other than RMA transfers, Late Complete occurs.

- *Early Fence*: It is the wait time associated with an epoch-closing fence call that occurs before the RMA transfers complete. Early Fence could impact both origin and target processes in a fence epoch.

- *Wait at Fence*: A closing fence call in any process must block until the same call occurs in all the processes in the group over which the RMA window is defined. If any process delays its call to fence while it has no or no more RMA communication to transfer, then it inflicts a Wait at Fence inefficiency to the other processes that issue their epoch-closing fence earlier for the same RMA window. Wait at Fence is a superset of Early Fence.

There is another inefficiency pattern not documented in [38, 55]. Since a passive target epoch does not involve an explicit epoch from the target side, situations of inefficient latency transfers are less obvious. However, when one considers at least two origins, we can define a new inefficiency pattern inflicted by current lock holders to subsequent lock requesters. That inefficiency that we trivially name ***Late Unlock*** can occur if:

1. The current holder possesses the lock exclusively. All the RMA transfers of the epoch have already completed but the holder delays the call to `MPI_WIN_UNLOCK` while there are any number of requesters willing to acquire the same lock (exclusively or not).

2. The current holders possess the lock in a shared fashion. All the RMA transfers are completed for all the holders at a time t. At least one holder is holding the lock beyond t while an exclusive lock requester is waiting.

Which inefficiency pattern can occur in an application might depend on the MPI implementation it is using. Actually, the MPI standard does not specify the blocking or nonblocking nature of most epoch-opening synchronizations. The specification simply requires RMA operations to avoid accessing non-exposed remote epochs; and epoch-closing routines not to exit until all the RMA transfers originating from or directed towards the epoch are done transferring; at least locally. In practice however, most modern and major MPI libraries [69, 71, 79] provide nonblocking epoch-opening routines; and rightfully so, because the blocking design of those phases of one-sided communication is documented as suboptimal [7, 104]. As a consequence, most MPI libraries avoid incurring Late Post on `MPI_WIN_START` invocation. Late Post can still be incurred at `MPI_WIN_COMPLETE` though.

Even though MPI-2.2 did not strictly specify RMA communication calls as nonblocking, it is customary for implementations to only provide nonblocking versions of those routines for the same reasons described in [7, 104]. Thus, Early Transfer is generally avoided altogether. Additionally, MPI-3.0 has now explicitly specified the RMA calls as nonblocking; making the Early Transfer pattern inexistent as per the standard itself. Furthermore, Early Wait can be mitigated by the use of `MPI_WIN_TEST`. However, Late Post, Late Complete, Early Fence and Wait at Fence are still as much of a problem in MPI-3.0 RMA as they were in MPI-2.0 RMA.

One can notice that, with Early Transfer not being an actual issue, it is not useful to define a "Late Fence" inefficiency pattern to mirror Late Post in the case of fence epochs. In fact, all the inefficiency patterns harm by creating and propagating delays to some activity occurring after the immediately concerned epochs. If Early Transfer was an actual issue, then the delay would impact what happens inside the immediately concerned epochs as well. In the case of GATS, the Late Post impact is felt at the origin-side epoch-closing routine, that is, `MPI_WIN_COMPLETE`. Similarly, a Late Fence impact, if any, would be felt at the epoch-closing fences. All the fence-related issues felt at epoch closing can only lead to either Early Fence or Wait at Fence. Consequently, one cannot isolate a so-called Late Fence effects from those of either Early Fence or Wait at Fence.

5.2.2 Serializations in MPI One-sided Communications

MPI one-sided communications are always nonblocking. However, for a given process and RMA window, unless the RMA communications occur in the same epoch, they do not really provide the benefits of nonblocking messaging. In Figure 5.1 for instance, any communication such as RMA_{b_0} that belongs to the second epoch is strictly serialized with any communication such as RMA_{a_0} that belongs to the first epoch. However, an MPI process cannot be constrained to issuing all its RMA communications inside a single epoch; especially if it aims to communicate with disjoint sets of targets.

In a two-sided-based application, unless data dependencies dictate otherwise, a performance-conscious programmer always opt for the approach in Figure 5.2(b). The obvious advantage of this nonblocking scheme is the parallel transfer of the communications; with a potentially reduced overall latency . Additionally, if the transfer of any `MPI_IRECV` suffers from late peer arrival, the resulting latency does not propagate to any of the subsequent communication call.

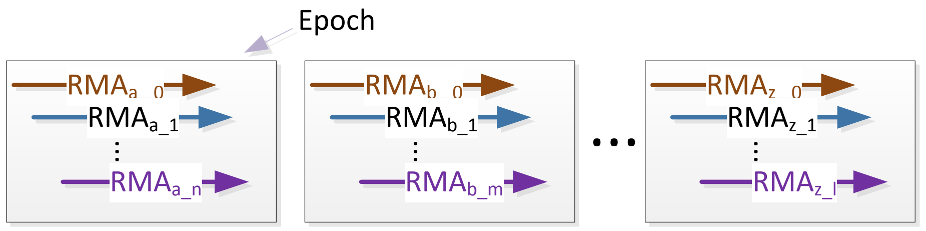
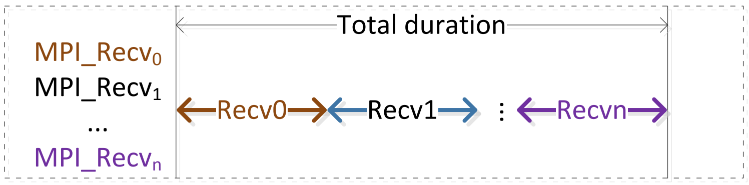
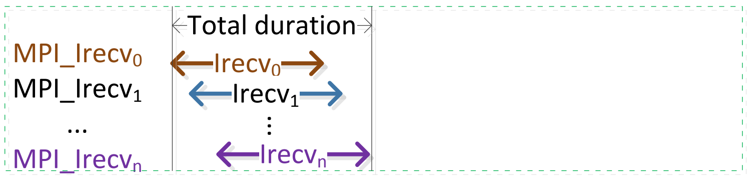


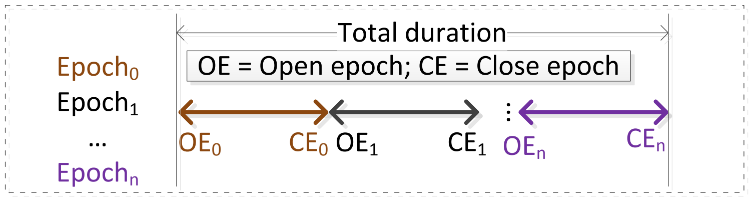
Figure 5.1: Parallel RMA in serialized epochs



(a) Blocking two-sided



(b) Nonblocking two-sided



(c) One-sided

Figure 5.2: Serialized vs. non-serialized communications

In the scheme presented in Figure 5.2(a), late peer arrival consequences propagate in chain to the other operations inside the receiver; and to all the senders of each of the subsequent `MPI_RECV` as well if the communications happen to resort to the Rendezvous protocol. The blocking scheme of Figure 5.2(a) is obviously very inefficient; and the programmer opts for it only when it is unsafe to do otherwise. Every time one-sided communications resort to multiple epochs, the current specification of MPI RMA (Figure 5.2(c)) confines the programmer into an equivalent of the communication scheduling shown in Figure 5.2(a) (See Section 5.4.2). As shown in Figure 5.1, two RMA communications in two distinct epochs of the same process fall into the scenario of Figure 5.2(c).

Serial epochs limit the ability of the HPC application programmer to express parallelism. There is no reason why epochs towards disjoint sets of targets should be serialized if the programmer knows with certainty that all the RMA writes touch different memory regions. There is in fact no justification to the ***unavoidable*** serialization of any epoch at all. The need for massive concurrency in the current petascale and the upcoming exascale computing eras requires MPI to leave the serialization decisions to the HPC application programmer; just as it is already the case for two-sided and collective communications.

5.3 Opportunistic Message Progression as Default Advantage for Nonblocking Synchronizations

To visualize a generic benefit of parallel epochs, one can map the MPI API calls made by the HPC application to the actions fulfilled by the MPI middleware. The MPI progress engine does not necessarily trigger a communication that has been initiated. Just like the Rendezvous protocol discussed in Chapter 3, many communications require preconditions to be met for their data transfers to start. A simple precondition in the case of passive target one-sided communications for instance could be the availability of remote lock.

When a precondition is not fulfilled, an initiated communication is kept inside the MPI middleware in a pending state. Figure 5.3 shows c_0 , c_1 and c_2 as examples of communications that were pending in the MPI middleware. In order to ensure that communications whose preconditions were not met at the time they were initiated still get a chance to complete before their equivalent completion detection calls, the MPI progress engine behaves in an *opportunistic* way. Every time it gets a chance to use the processor for a communication c_i for instance, the progress engine tries to progress all previously pending communications whose preconditions were met in the meantime. For instance, Figure 5.3 shows the progress engine leveraging the CPU time of communication c_3 to progress communications c_0 and c_2 .

Similarly, the pending communication c_4 is progressed when communication c_5 is issued.

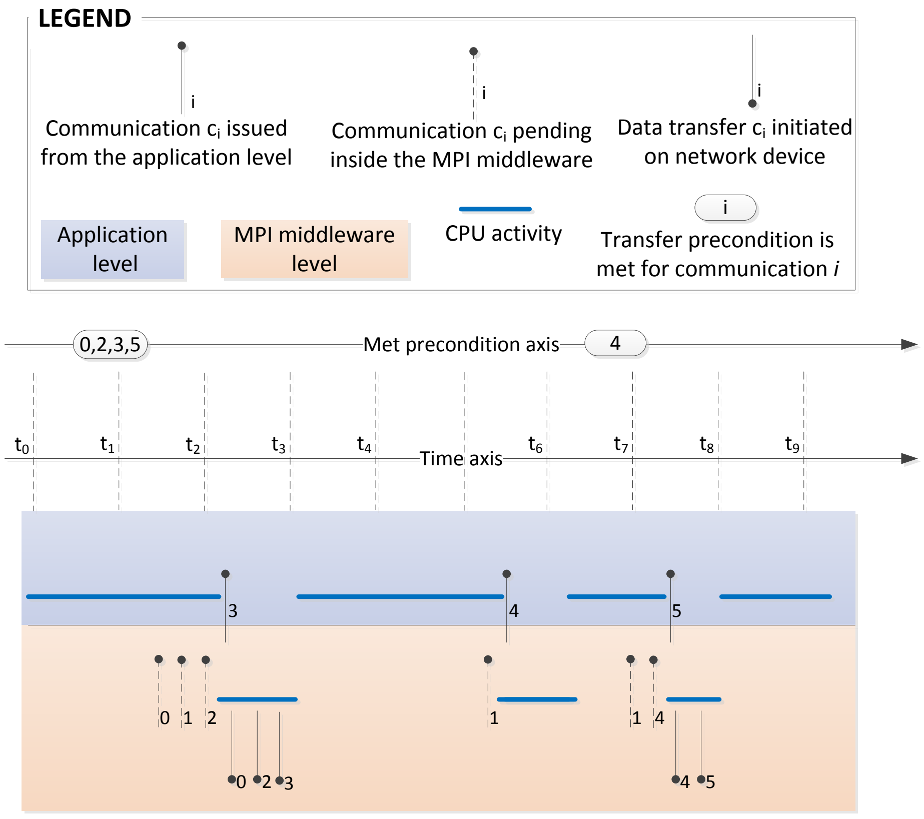

LEGEND
i
Communication c_i issued from the application level
i
Communication c_i pending inside the MPI middleware
i
Data transfer c_i initiated on network device
Application level
MPI middleware level
CPU activity
i
Transfer precondition is met for communication i
0,2,3,5
Met precondition axis
4
t0
t1
t2
t3
t4
Time axis
t6
t7
t8
t9
3
4
5
0 1 2
1
1 4
0 2 3
4 5

Figure 5.3: Opportunistic message progression

One can make a few important observations from how application-level activities map to their middleware-level equivalent and how the CPU is shared between the two levels. In Figure 5.3 for instance, the precondition 5 was fulfilled since t_1 on the time axis; but communication c_5 had to wait until about t_8 to start transferring. Similarly, communication c_4 could not start right after it was initiated between t_5 and t_6 because its precondition is fulfilled later at about t_7 . For any communication with initiation time t_c and precondition realization time t_{pc} , the earliest transfer starting time t_{et} is:

$$t_{et} = max(t_c, t_{pc}) \tag{5.1}$$

A communication which completes early allows both the local and involved remote peers to move on earlier and minimize wait times. Thus, as much as possible, t_{et} should be minimized.

As a result, both t_c and t_{pc} must be minimized. An Epoch closed with blocking synchronizations does not allow the HPC programmer to minimize t_c in subsequent epochs. Incidentally, serial epochs make a suboptimal use of the opportunistic behaviour of the progress engine because they don't leverage early precondition fulfillment times, just like communication c_5 in Figure 5.3. In fact, replacing each communication in Figure 5.3 by an epoch, as shown in Figure 5.2(c), leads to the complete absence of pending items inside the progress engine. As a result, no transfer is ever opportunistically triggered, without respect to the currently realized preconditions.

The progress engine can only trigger the transfers that it is aware of; that is, those that have transitioned from the application level to the middleware level. As soon as any preventing condition is removed at application level, any communication is always better off pending inside the middleware; without respect to when its preconditions are fulfilled. Therefore, the latency propagation-minimizing version of the situation described by Figure 5.1 is the one depicted in Figure 5.4 where parallelization can increase the number of pending epochs inside the middleware; and incidentally the effects of opportunistic progression.

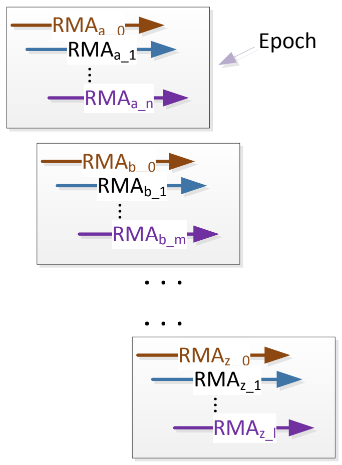


Figure 5.4: Parallel epochs

5.4 Impacts of Nonblocking RMA Epochs

Nonblocking epochs can be fulfilled with the nonblocking handling mechanism that already exists in MPI (Section 2.1.5). Epoch closing, opening and flushing routines would first be initiated in a nonblocking way; and then completed with any of the ***test*** or ***wait*** family of routines.

With most HPC network devices being RDMA-enabled, one obvious advantage of having nonblocking communications is an increased level of communication/computation overlapping. In the case of nonblocking epochs, a few additional positive impacts are described in the following subsections.

5.4.1 Fixing The Inefficiency Patterns

We mentioned in Section 5.2 that the Late Post, Late Complete, Early fence, Wait at Fence and Late Unlock inefficiency patterns are yet to be fixed. All five inefficiency patterns find a sound solution with nonblocking epochs.

Late Post

We define t_0 as the invocation time of the origin-side epoch-closing routine. We assume that `MPI_WIN_POST` is late and occurs D_p after t_0. The data transfer duration of the RMA communications in the epoch is D_{tr}; and the time needed to close the origin-side epoch if all transfers are already completed is ε. In these conditions, the earliest the next origin-side activity can start after `MPI_WIN_COMPLETE` is:

$$t_{1\_blocking\_epoch} = t_0 + D_p + D_{tr} + \varepsilon \tag{5.2}$$

With a nonblocking origin-side epoch closing, the RMA transfer duration does not propagate to the next activity. More importantly, any delay created by the target not exposing its epoch on time does not propagate to the next origin-side activity. Consequently, the earliest the next activity can start becomes:

$$t_{1\_nonblocking\_epoch} = t_0 + \varepsilon \tag{5.3}$$

Equation 5.3 shows that nonblocking epochs allow the origin process to completely mitigate the Late Post situation if the next activity does not depend on data produced in the epoch. Then, in absence of data dependency, since the next activity starts sooner than $t_{1\_blocking\_epoch}$, it overlaps at least partially with the delay and RMA transfer of the previous epoch even if it occurs after the epoch is over; allowing the overall completion of both activities to be reduced. Needless to mention that the earliest starting time of the next activity determines the earliest starting time of any other subsequent activity.

Early Fence

We define t_0 here as the time when MPI_WIN_FENCE or its nonblocking equivalent is invoked. Using the previous definitions for all the other relevant time variables, the earliest moment the next activity can start with a blocking fence is:

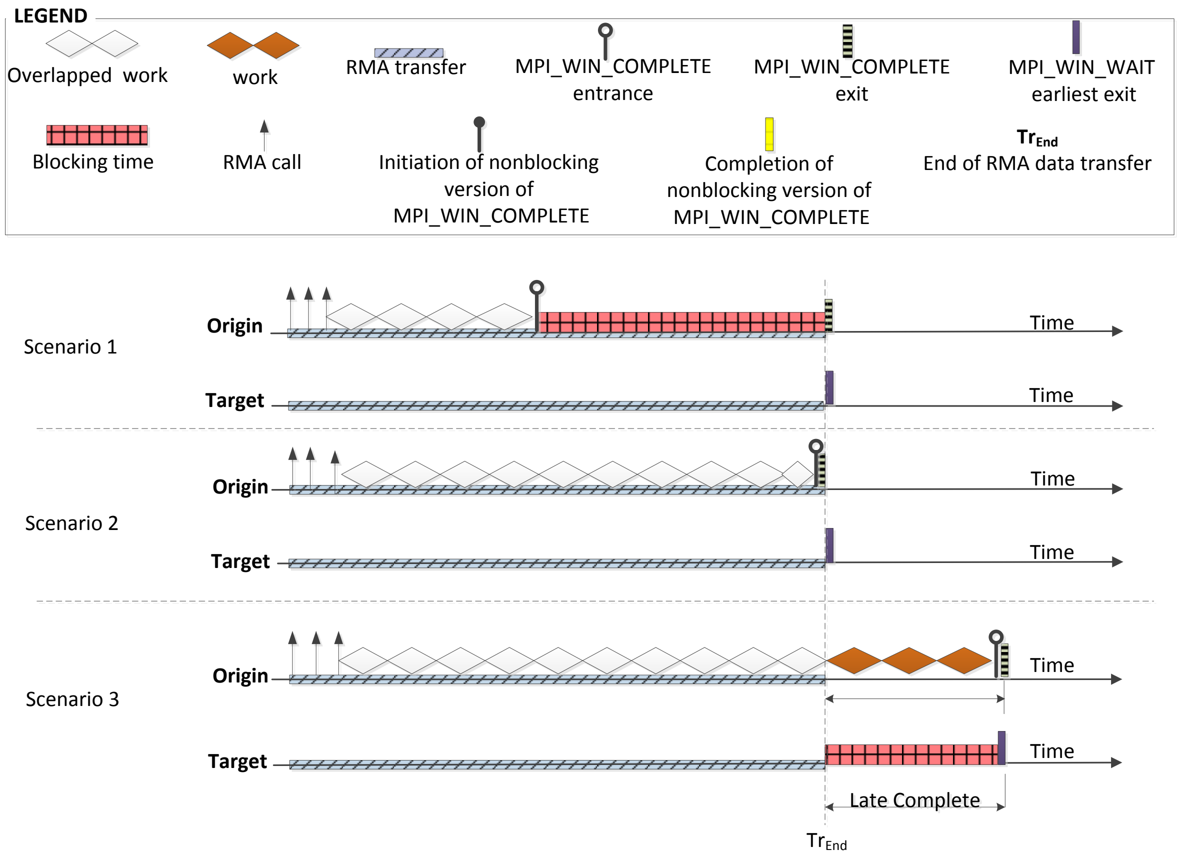
$$t_{1\_blocking\_epoch} = t_0 + D_{tr} + \varepsilon \tag{5.4}$$

With a nonblocking version of MPI_WIN_FENCE, the next activity can also start at $t_{1\_nonblocking\_epoch}$ as expressed by Equation 5.3; with the same positive consequences mentioned above.

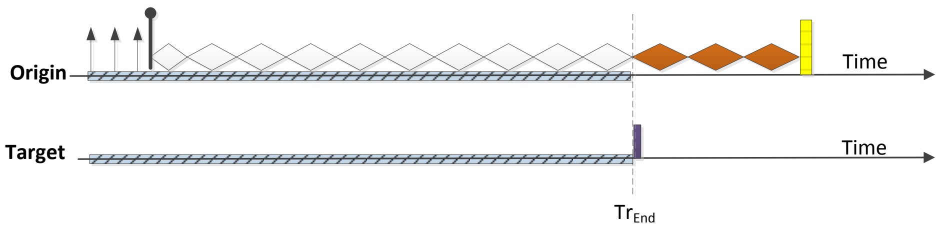
Late Complete

Blocking epochs offer no clear winning strategy as to the right synchronization call timing to maximize performance. Actually, they offer conflicting strategies for latency avoidance or mitigation. Epochs are critical section-like regions; and as such, they should be kept as short as possible (scenario 1 of Figure 5.5(a)). At the same time, CPU idling should be avoided. Therefore, useful work should be overlapped with the communication of an epoch if that communication is known to have a somehow lasting transfer time (scenario 3 of Figure 5.5(a)). The performance-savvy MPI programmer resorts to this second approach to avoid the CPU idleness of scenario 1 in Figure 5.5(a). It is unrealistic to expect the occurrence of scenario 2 of Figure 5.5(a) because the application cannot calibrate its work length to be exactly the length of its data transfer.

The critical section thinking means that MPI_WIN_COMPLETE is invoked as quickly as possible. For an access epoch, there is even no guarantee that the corresponding exposure epoch will be opened on time. In a GATS setting, this means that an early blocking access epoch-closing call increases both the risk and magnitude of Late Post suffering by origin processes. As a reminder, the origin-side epoch closing routine blocks as long as the RMA transfers do not complete. However, as long as the target epoch is not open, the RMA transfer cannot even start.


LEGEND
Overlapped work
work
RMA transfer
MPI_WIN_COMPLETE entrance
MPI_WIN_COMPLETE exit
MPI_WIN_WAIT earliest exit
Blocking time
RMA call
Initiation of nonblocking version of MPI_WIN_COMPLETE
Completion of nonblocking version of MPI_WIN_COMPLETE
Tr_{End} End of RMA data transfer
Scenario 1
Origin
Target
Time
Scenario 2
Scenario 3
Late Complete
Tr_{End}

(a) Latency transfer tradeoff leading to Late Complete



(b) Solving Late Complete with nonblocking version of `MPI_WIN_COMPLETE`

Figure 5.5: Impact of nonblocking epochs on Late Complete

The Late Complete inefficiency results from the hunt for communication/computation overlapping; and could therefore be the consequence of applying recommended HPC programming practices. The situation leading to Late Complete (scenario 3 in Figure 5.5(a)) is also a selfish attempt made by the origin process to avoid stalling while its own RMA transfer are in progress. By doing so, the origin process is better off; but it potentially transfers an unjustified wait time to the target. The alternative is the critical section thinking which, as shown in scenario 1 of Figure 5.5(a) guarantees the absence of the Late Complete inefficiency pattern at the expense of the origin process. The two (realistic) scenarios 1 and 3 are therefore the two aspects of an unavoidable tradeoff where one peer potentially suffers some undesirable wait. With a nonblocking version of `MPI_WIN_COMPLETE` (Figure 5.5(b)), the tradeoff disappears because it becomes possible to keep the origin in scenario 3 while simultaneously having the target in scenario 1 of Figure 5.5(a). The origin can adopt a critical section thinking by closing the epoch as soon as possible; avoiding any unjustified wait to the target. At the same time, since the RMA transfer continues even after the epoch is closed in a nonblocking way, the origin can both avoid stalling and also overlap some work with the communication progression.

Wait at Fence as the Risky Remedy for Early Fence

Early fence and Wait at Fence are the two unfortunate options of a blind strategy decision making. In order to avoid Early fence with blocking epoch routines, a process should issue its fence call later. Unfortunately, by doing so, it might inflict Wait at Fence to the other participant processes. Once again, nonblocking epoch routines allow every participant to be selfish without inflicting any inefficiency to the other participants. With the right design in place, the nonblocking fences are issued early for every participant; and then the subsequent calls to ***test*** or ***wait*** can be delayed as much as needed without any latency transfer to remote peers.

Late Unlock

Late Unlock can be analyzed with Figure 5.5 by replacing the "origin" with the "current lock holder" and the "target" with the "subsequent lock requester". There is no incentive for the current lock holder to issue `MPI_WIN_UNLOCK` early; as it might experience the same stalling as the origin in scenario 1 of Figure 5.5(a). Unfortunately, by putting itself in scenario 3 of Figure 5.5(a), the current lock holder could inflict an undesirable wait time to a subsequent lock requester. Just as in the case of Late Complete, a nonblocking version of `MPI_WIN_UNLOCK` completely voids the painful tradeoff.

5.4.2 Unlocking or Improving New Use Cases

Working Around the Impossible "Epoch Inside Epoch" Issue

The possibility of opening an epoch without waiting for the completion of the previous one is a powerful feature. To explain how nonblocking epochs allows new use cases, we define two distinct groups G_0 and G_1 of target processes. Many reasons could require the creation of distinct groups for disjoint sets of targets. For instance, a group might accept an optimization (e.g. `MPI_MODE_NOPUT` in matching target epoch) while the other one could not. We also define r_{j_Gi} as the j^{th} RMA communication towards a target in G_i. With a single RMA window w, it is not allowed for an origin process to issue the sequence of calls:

$$\texttt{MPI\_WIN\_START}(G_0, w), r_{0\_G0}, r_{1\_G0}, ..., r_{k\_G0},$$

$$\texttt{MPI\_WIN\_START}(G_1, w), r_{(k+1)\_G1}, r_{(k+2)\_G0} \cdots,$$

$$\texttt{MPI\_WIN\_COMPLETE}(w), \texttt{MPI\_WIN\_COMPLETE}(w)$$

One obvious problem is that it is not clear which invocation of `MPI_WIN_COMPLETE` closes the epoch over G_0 or G_1. The only way to have those two distinct access epochs from the same origin process is to serialize them; i.e., close the one opened over G_0 before opening the one over G_1. However, as mentioned earlier, blocking epoch serialization is a hindrance for both performance and scalability. The other solution is to perform each of the epochs on distinct RMA windows. Unfortunately, this second approach works only for reasonably structured execution flows in which the number of epochs can be known ahead of starting the computational loops which perform the RMA operations. If the execution flow is very unstructured instead; and the amount of required RMA communications and epoch characteristics are determined by runtime dynamic conditions, resorting to multiple RMA windows could be at best inefficient; if not impossible. Creating the RMA windows on demand introduces barrier-like collective calls in the middle of the computations; with a potentially increased effect of late peer arrival. Sometimes, such an approach could even be difficult to orchestrate without deadlock risks.

Even if the MPI programmer can guess an upper limit of the number of required RMA windows, statically creating them all ahead of time is a poor workaround, especially if the program can require a large number of those windows.

With nonblocking epochs, the same origin process can use a unique RMA window to issue as many epochs as possible without blocking. It then becomes possible to fulfill the goal of "epoch inside epoch" in two different ways. In the first approach, the two epochs over G_0 and G_1 would still be issued serially. The epoch E_1 over G_1 is not issued until the epoch E_0 over G_0 is entirely issued; but thanks to nonblocking synchronizations, E_1 does not wait for the completion of the data transfer of E_0 before being issued. Instead of having epochs inside epochs, the application realizes epoch data transfers occurring inside previous epoch data transfers; another way of reaching the same goal in terms of overall latency. In a second approach, the application could break the epoch over G_0 into two epochs E_0 and E_2 as follows:

- Open an epoch E_0 over G_0.
- When the need arises to open the epoch E_1 over G_1 while E_0 is still open, close E_0 in a nonblocking fashion and open E_1. Issue the RMA calls of E_1 and close it in a nonblocking fashion.
- If the need arises to do additional RMA transfers over G_0 again, the application can simply open another epoch E_2 over G_0 again. By doing so, the application effectively realizes the equivalent of the "epoch inside epoch" scenario that is not possible with blocking synchronizations.

Improving the Inefficient Massively Unstructured Atomic Communication Pattern

Algorithms whose communication patterns perform massive transactions can be challenging to realize efficiently in MPI. The communication pattern goes as follows. At any given time, a set of peers $\{P_i\}$ can update a bunch of other peers $\{P_j\}$. $\{P_i\}$ and $\{P_j\}$ are not disjoint sets; any configuration is possible. Processes do not know ahead of time how many updates they will get; nor can they determine where these updates will originate from or what addresses they will target. As a consequence, it is very inefficient to resort to two-sided communications where the updates would require matching receives. Finally, the updates must be atomic; meaning that no one-sided epoch type but ***exclusive*** `MPI_WIN_LOCK` is adequate. We define:

- S as the set of processes.
- n as the number of times a process performs updates.
- i as the rank of the target process currently being updated.
- u as an atomic update operation. An update is an `MPI_ACCUMULATE`.

`MPI_ACCUMULATE` guarantees atomicity only when it is invoked with a predefined datatype for the target; and a single item of such datatype is being accumulated. If multiple predefined datatype elements must be accumulated, atomicity is guaranteed only on a per-element basis. As a transaction can accumulate more than a single element per update, it requires an exclusive lock to guarantee atomicity. Thus, we also define `MPI_WIN_LOCK`(`MPI_LOCK_EXCLUSIVE`,i), `MPI_WIN_UNLOCK`(i) as the calls which respectively lock exclusively and unlock the process of rank i. A single transaction is:

$$U_i = \texttt{MPI\_LOCK\_EXCLUSIVE}(\texttt{MPI\_LOCK\_EXCLUSIVE}, i), u, \texttt{MPI\_WIN\_UNLOCK}(i)$$

The communication pattern from the point of view of each process is therefore a large number of repetitions of the sequence U_i. For two distinct updates, even if they are directed towards the same target, they must occur in two distinct epochs; otherwise, they are no longer guaranteed to be atomic. As a result, the number of epochs generated by a process is strictly equivalent to the number of transactions that it performs. With blocking epochs, this communication pattern could perform poorly because every single one of the massive number of transactions is strictly serialized.

However, one can notice for any given origin process that two transactions towards two distinct targets need not be serialized because atomicity matters only for multiple transactions modifying the same data. For a given process, if we define U_{i_k} as the k^{th} transaction towards target i, then all the updates in the sequence U_{a_0} U_{b_0} U_{c_0} U_{a_1}, U_{d_0} can occur in parallel except for U_{a_0} and U_{a_1}. The current specification of MPI-3.0 RMA epochs serializes them all at the application level. In comparison, nonblocking epochs allows them all, including the two that are meant for target a, to be issued simultaneously. Then, because the lock of process a is obtained exclusively anyways, the progress engine takes care of serializing U_{a_0} and U_{a_1} internally; voiding any hazard associated with those two updates being concurrently active. In the meantime, all the other updates are issued without delay at application level.

5.5 Nonblocking API

For each potentially blocking epoch function `MPI_WIN_FUNC(LIST_OF_PARAM)`, a nonblocking version of the form `MPI_WIN_IFUNC(LIST_OF_PARAM,REQUEST)` is provided. `LIST_OF_PARAM` is the list of parameters of the blocking version while `REQUEST` is an output parameter used to detect completion as described in Section 2.1.5.

The nonblocking epoch-opening API is composed of `MPI_WIN_IPOST`, `MPI_WIN_ISTART`, `MPI_WIN_IFENCE`, `MPI_WIN_ILOCK` and `MPI_WIN_ILOCK_ALL`. While modern MPI libraries tend to provide nonblocking epoch-opening routines [7, 69, 71, 79, 104], the MPI standard is not specific about the blocking or nonblocking nature of all these functions; meaning that their behaviour is implementation-dependent. The API provided in this section enforces the uniform ambiguity-free nonblocking nature of its functions. For instance, while a blocking implementation of `MPI_WIN_START` is as standard-compliant as a nonblocking implementation, `MPI_WIN_ISTART` can only be nonblocking. We emphasize that `MPI_WIN_POST` was already specified as nonblocking in MPI-3.0; as a consequence, `MPI_WIN_IPOST` is provided solely for uniformity and completeness.

The nonblocking epoch-closing API is composed of `MPI_WIN_IWAIT`, `MPI_WIN_ICOMPLETE`, `MPI_WIN_IFENCE`, `MPI_WIN_IUNLOCK` and `MPI_WIN_IUNLOCK_ALL`. As a reminder, `MPI_WIN_FENCE` is used for both epoch opening and epoch closing. It also helps to remind that MPI-3.0 already provides `MPI_WIN_TEST` as the nonblocking equivalent of `MPI_WIN_WAIT`. However, the new `MPI_WIN_IWAIT` that we propose remains relevant. In fact, compared to `MPI_WIN_TEST`, the combination of `MPI_WIN_IWAIT` with the ***test*** family of nonblocking handling is more powerful because it allows the asynchronous and wait-free initiation of subsequent epochs. `MPI_WIN_TEST` detects the completion of the current exposure epoch in a nonblocking manner;

but since no other exposure epoch can be opened until the completion actually occurs, it does not prevent application-level epoch serialization. The only merit of `MPI_WIN_TEST` is to prevent the CPU from idling while waiting for the currently active exposure epoch to complete.

Finally, the nonblocking ***flush*** API is composed of `MPI_WIN_IFLUSH`, `MPI_WIN_IFLUSH_LOCAL`, `MPI_WIN_IFLUSH_ALL` and `MPI_WIN_IFLUSH_LOCAL_ALL`. With these new routines, a passive target epoch can remain entirely nonblocking even if it hosts some function calls from the ***flush*** family.

5.6 Semantics, Correctness and Progress Engine Behaviours

In general, dealing with a communication which completes exactly at the call site has the advantage of allowing the programmer to reason about the exact time when a buffer is modified at application level. In comparison, a nonblocking communication only allows the programmer to reason about "the latest moment" when a buffer is done being modified. Between the moment the nonblocking communication call exits and when the buffer safety guarantees are provided, there is a window of application-level instructions which leaves place to various kinds of data corruption hazards. In the specific case of one-sided communications, the closing synchronization calls are meant to guarantee memory consistency as far as the buffers touched by RMA are concerned. Thus, if the closing synchronization ceases to provide that guarantee because of its nonblocking nature, the programmer is left with a tool which could be seen as potentially error-prone. Nevertheless, one can make a case for such a tool by considering the options of 1) giving the HPC programmer performance-restricted features meant to prevent him from making mistakes or 2) powerful tools meant for the informed HPC programmer. It helps to point out that the second option is not unusual in supercomputing and in MPI for that matter. The only requirement when that option is chosen is to clearly spell out the semantics, correctness and hazardous situations that might be inherent to the tool being given to the programmers. This requirement, that we fulfill in this section, is an integral part of the novel proposal.

For an epoch, we distinguish between *application-level lifetime* and *internal lifetime.* The beginning and ending of the application-level lifetime respectively correspond to the application-level opening and closing of the epoch. We use the terms “open” and “closed” to define the boundaries of the application-level lifetime of an epoch. The internal lifetime takes place inside the middleware; it starts when the epoch is internally *activated* for progression by the progress engine; and it ends when the epoch is done being progressed and all the internal closing notifications have been sent to all the relevant peers. The terms “activated” and “completed” define the boundaries of the internal lifetime of an epoch. An epoch that is still alive internally is said to be *active*.

For RMA synchronizations as defined by the MPI-3.0 standard, application-level and internal lifetimes occur strictly at the same time; and the distinction is not even meaningful. With the nonblocking synchronizations being proposed in this dissertation, the two lifetimes must be distinct because they do not necessarily occur at the same time. In fact, the most important concept that distinguishes blocking and nonblocking epoch closings is the distinction between “closing” and “completion”. An epoch closed with a blocking synchronization always completes when the synchronization routine exits. In comparison, an epoch closed with a nonblocking synchronization might not complete until completion is explicitly detected with one of the ***test*** or ***wait*** family of functions. Figure 5.6 shows the distinction between the two levels of lifetime.

We designate by *deferred epoch* an epoch that cannot be activated as soon as it is opened at application level. In Figure 5.6, all the application-level epochs but e_2 lead to deferred epochs. A deferred epoch is different from an epoch that is not yet granted access. In fact, all kinds of epochs can be deferred; including exposure epochs for which access acquisition is not meaningful.

5.6.1 Semantics and Correctness

The semantics of nonblocking epochs complements the existing blocking epochs of the MPI 3.0 specification. Additionally, it modifies the completion conditions to account for the split-phase wait of nonblocking epoch-closing. The correctness specification describes conditions for the nonblocking synchronizations to guarantee the same level of safety as the existing blocking synchronizations. The correctness rules also cover the hazard-free practices for creating epochs which exist in overlapping time frames. Finally, the rules spell out the behaviour to expect from the progress engine when nonblocking synchronizations are used. We define the following application-level events and entities:

- OE_i: Open the i^{th} epoch in a potentially blocking fashion.
- IOE_i: Open the i^{th} epoch in a deterministically nonblocking way.
- CE_i: Close the i^{th} epoch in a blocking way.
- ICE_i: Close the i^{th} epoch in a nonblocking way.
- r_k: Issue the k^{th} RMA communication.
- AE_i: The i^{th} access epoch.
- E_i: The i^{th} epoch. The access or exposure nature of E_i is not of importance.

In the rules that follow, only the last of a succession of epochs is potentially still open. Thus, unless otherwise specified, the succession of nonblocking epochs E_i, E_{i+1} never means that E_i and E_{i+1} overlap at application level. In fact, any series (E_i, E_{i+1}) is equivalent to either $(IOE_i, ICE_i, IOE_{i+1}, ICE_{i+1})$ if both epochs are already closed; or $(IOE_i, ICE_i, IOE_{i+1})$ if the last of both epochs is still open. The following rules, meant to define constraints, clarifications and recommendations, apply:

1. RMA communications happen inside nonblocking epochs with the exact same rules and constraints they are currently subject to in the MPI-3.0 specification. The situations that were specified as *undefined* and *erroneous* in the MPI-3.0 standard still hold with the addition of nonblocking epochs.

2. For an epoch which is closed in a nonblocking fashion, there is a clear distinction between the end of the epoch which is commanded by a call to ICE; and its completion which requires a call to ***test*** or ***wait*** functions. For instance, in Figure 5.6, epoch e_0 is closed before the time step t_1; but its completion at application level occurs at t_4. The

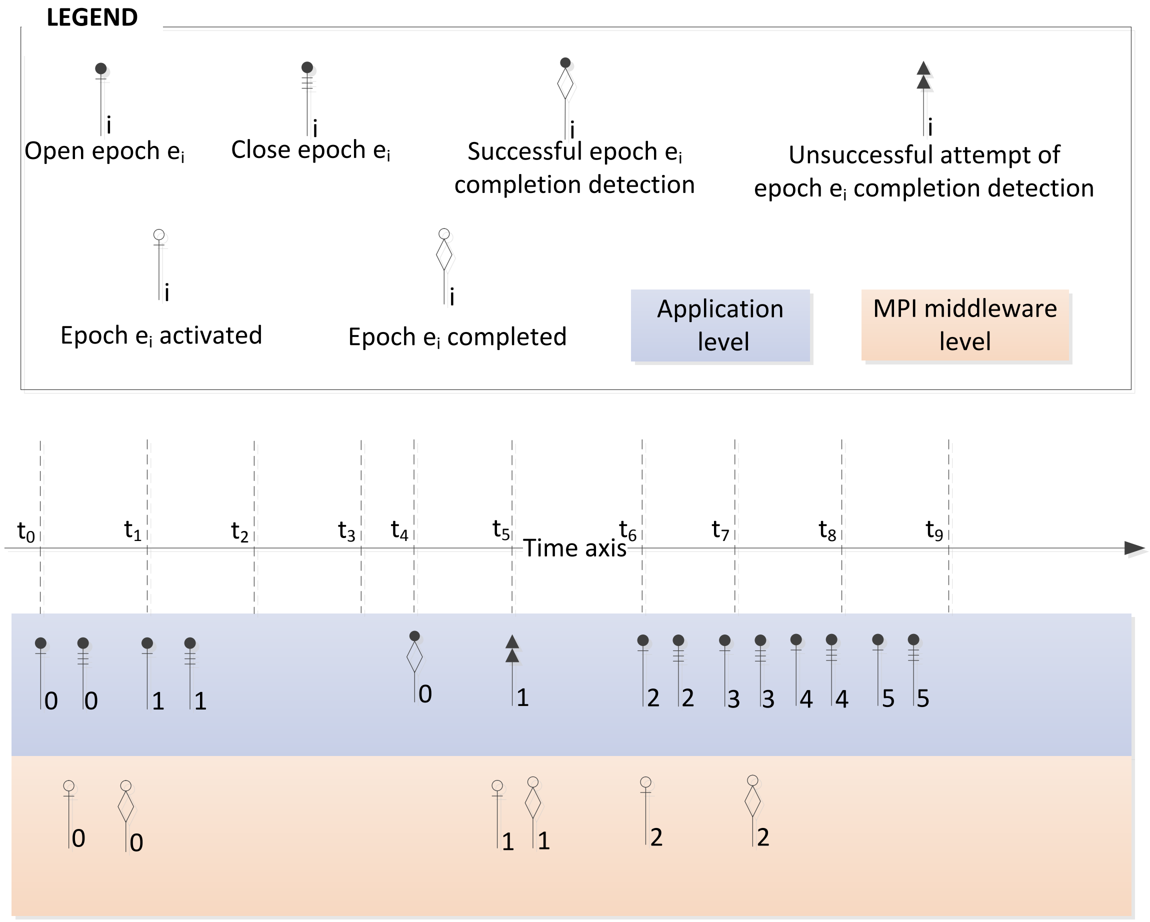


Figure 5.6: Mapping of application-level nonblocking epochs to MPI middleware-level handling

access to the buffers involved in the RMA communications of an epoch which is closed in a nonblocking manner remains unsafe until completion is successfully detected. In Figure 5.6, the application can assume that it is safe to modify the buffers touched by epoch e_0 only after t_4. However, the buffers touched by e_1 are not safe to modify even at t_5; because the completion detection attempt is not successful. One can notice the contrast with blocking epoch-closing synchronization routines which always guarantee completion and buffer safety by the time they exit.

3. Any issued RMA communication belongs to the most recently opened access epoch. For instance, in the sequence $(IOE_0, r_0, r_1, ..., r_k,\ ICE_0, IOE_1, r_{k+1}, r_{k+2},\ ..., r_{k+m}), r_{k+1}$ to r_{k+m} belong to AE_1 even if AE_0 has not completed yet.

4. It is possible to open an epoch in a nonblocking fashion and close it in a blocking fashion and vice versa. In fact any combination of blocking and nonblocking routines can be used for the synchronization routines that make an epoch.

5. An epoch is guaranteed to be entirely nonblocking only if its opening and its closing are all nonblocking. If it is a passive target epoch, any ***flush*** call made inside the epoch must be nonblocking as well for the epoch to remain nonblocking.

6. The use of nonblocking routines to open, close or flush an epoch does not impact the ordering of its RMA operations. As a consequence, blocking and nonblocking epochs have the exact same intra-epoch ordering and atomicity rules. These rules are the ones currently defined in the MPI-3.0 specification.

7. In active target, access and exposure epochs match one another in a FIFO manner. The oldest deferred access epoch is always the next to match the oldest deferred exposure epoch, and vice versa. If any of the sides does not have deferred epochs, then the match occurs between the oldest deferred epoch on one side and the next opened epoch on the other side. This rule maintains the possibility of reasoning about the deterministic write ordering when multiple nonblocking epochs are used over the same memory region. The determinism of reasoning applies to both implementers and the end-users.

8. For a given process, epochs are always activated serially in the order where they become pending inside the progress engine. As a consequence, if for any reason E_k cannot be activated yet, then E_{k+1} cannot be activated either. This rule means that epochs are not skipped. This rule only defines activation constraints and does not mean that E_k must be internally completed before E_{k+1} is activated.

9. `MPI_WIN_IFENCE` entails a barrier semantics wherever `MPI_WIN_FENCE` does; that is, whenever the fence call ends an epoch. In particular, if a call to `MPI_WIN_IFENCE` must close epoch E_k and open epoch E_{k+1}, the progress engine of each process must internally delay activating E_{k+1} until it receives the E_k completion notifications from all the other peers encompassed by the RMA window. The delay impacts how long it takes for completion

to be fulfilled for `MPI_WIN_IFENCE` but it must produce no noticeable blocking call at application level.

10. ***Test*** or ***wait*** functions must always eventually be invoked at application level to detect the successful completion of each request, even if the completion of the request can be inferred by other means. A Failure to do so leads to resource leaks. In particular:
    (a) If the completion of a passive target epoch is noticed at application level via a ***test*** or ***wait*** function, requests associated with all the ***flush*** calls issued inside the epoch are of course guaranteed to have completed as well. Nevertheless, if not yet invoked, ***wait*** or ***test*** routines must still be called on these flush requests to clean up their associated internal objects.
    (b) By the time `MPI_WIN_FREE` returns, all the epochs associated with the concerned RMA window are guaranteed to have completed. Nevertheless, ***wait*** or ***test*** calls are still required for each request whose completion detection has not yet been performed at application level.

11. All the aforementioned rules define the dynamics of epoch management for a given RMA window. When multiple RMA windows expose overlapping memory regions, the current MPI 3.0 specification rules for multiple RMA windows apply. It is the application programmer's responsibility to provide means for memory consistency hazard avoidance when multiple RMA windows have simultaneous epochs that alter overlapping ranges of virtual addresses.

For an MPI one-sided communication programmer who already has some experience with writing nonblocking communication code with either two-sided or collectives routines, the new nonblocking synchronizations add a very reasonable level of complexity. The actual complexity of this proposal lies in the realization and implementation inside the middleware. In fact, rules 1 and Rule 11 state constraints or recommendations that already existed in the MPI-3.0 specification for one-sided communications. They simply establish the lack of change of those recommendations or constraints with the introduction of nonblocking synchronizations. Rule 3 is a clarification. Rule 4, Rule 5 and Rule 6 are also clarifications; but they also explain various kinds of flexibility that are offered to the MPI application programmer with nonblocking synchronizations. The recommendation expressed by Rule 10 is the equivalent for nonblocking synchronizations of a recommendation that already exists in MPI. Requests must be explicitly freed in MPI in all circumstances; even if they can be inferred to have completed. The constraints expressed by Rule 7, Rule 8 and Rule 9 are entirely implementation-related. These rules also serve as clarification for what should be expected at application level.

Rule 2 is the only new correctness burden put on the application programmer. For a given RMA window, if Rule 2 is respected, then any epoch or succession of epochs built with nonblocking synchronizations can be rewritten with an epoch or a succession of epochs built with the existing MPI-3.0 synchronizations. Rule 2 means that if any load or store is allowed only after an epoch built with blocking epoch-closing synchronizations, then it is allowed only after the ***wait*** or successful ***test*** that completes the equivalent epoch built with nonblocking epoch-closing synchronizations. One can notice that Rule 2 is not any more complex than the constraint that forbids the access of a buffer manipulated by an `MPI_IRECV` or `MPI_ISEND` until their associated ***wait*** or ***test*** call completes successfully.

5.6.2 Info Object Key-value Pairs and Aggressive Progression Rules

In order to guarantee correctness by default, the progress engine does not activate an epoch while another one is still active. One can notice that this default behaviour does not defeat the purpose of nonblocking epochs or synchronizations. In fact, for any given RMA window, nonblocking synchronizations allow any combinations of multiple epochs to be simultaneously pending inside the progress engine. As a result, the epochs can be activated at the earliest possible time thanks to opportunistic message progression. In comparison, at application level, blocking synchronizations allow a maximum of one origin-side epoch and one target-side epoch for any given RMA window. As a consequence, blocking synchronizations lead to the following three suboptimal situations:

- Complete absence of opportunistic progression for a set of origin-side-only epochs.
- Complete absence of opportunistic progression for a set of target-side-only epochs.

- A maximum of one epoch being opportunistically progressed for any sets of mixed origin-side and target-side epochs.

Even though it surpasses blocking synchronizations, the default behaviour of nonblocking epochs can still be limiting in situations where the programmer knows of the absence of correctness hazards for a given target. Obviously, further optimizations are possible beyond the sole opportunistic message progression advantage. Thus, in addition to the API, we provide the following info object key-controlled boolean flags that the programmer can associate with an RMA window:

- `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER`: If its value is 1, then the progress engine can activate and progress any origin-side epoch even if an immediately preceding origin-side epoch is still active.
- `MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER`: If its value is 1, then a subsequent origin epoch can be activated and progressed concurrently with an immediately preceding and still active exposure epoch.
- `MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER`: If its value is 1, then the progress engine can activate and progress any target-side epoch even if an immediately preceding target-side epoch is still active.
- `MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER`: If its value is 1, then a subsequent target epoch can be progressed concurrently even if an immediately preceding origin-side epoch is still active.

The consequence of any of these four flags being enabled is that the RMA communications of a subsequent epoch E_{i+1} can end up being transferred before those of a previous epoch E_i. If the epochs contain any `MPI_GET`, `MPI_RGET`, `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE` or `MPI_FETCH_AND_OP`, write reordering can occur in the origin address space with respect to the chronology of E_i and E_{i+1}. Similarly, if the epochs contain `MPI_PUT`, `MPI_RPUT`, `MPI_ACCUMULATE`, `MPI_RACCUMULATE`, `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE`, `MPI_FETCH_AND_OP` or `MPI_COMPARE_AND_SWAP`, write reordering can occur in some targets with respect to the chronology of E_i and E_{i+1}. In general, a read reordering in a target can lead to a write reordering in some origin. Write reordering is not a desirable outcome because it bears hazards. Justifiably, all these flags are disabled by default. It is assumed that the HPC programmer can guarantee, via the knowledge of the data access pattern of his application, that the RMA activities of concurrently progressed epochs involve strictly disjoint memory regions.

The flags set by these new info object key-value pairs all apply to a given RMA window and do not bear any meaning across distinct window objects; that is, the flags operate at window level and independently for each window. The flags allow the progress engine to perform aggressive message progression by completing epochs out-of-order if required. They lead to new semantics relaxation rules. However, we have to be cautious in order to 1) allow HPC programmers to be able to still reason about data modification outcomes and 2) avoid modifications that introduce ambiguity in the existing MPI-3.0 semantics. As a result, the progress engine optimization flags do not apply and have no effect on epochs opened by `MPI_WIN_LOCK_ALL`, `MPI_WIN_FENCE` and their respective nonblocking equivalents.

For `MPI_WIN_LOCK_ALL` and `MPI_WIN_ILOCK_ALL`, the reason for not supporting the progress engine optimization flags are as follows. The MPI specification does not allow a process to create an exposure epoch if it is already locked. Similarly, an already locked process cannot be exposed in GATS. These restrictions apply at application level and are therefore the responsibility of the HPC application programmer; the middleware does not have to enforce or check them. However, even if the programmer respects the constraint, `MPI_WIN_LOCK_ALL` or `MPI_WIN_ILOCK_ALL` could still reproduce the forbidden scenarios inside the middleware if we allow any of the four optimizations to apply to them. The middleware cannot be allowed to create a violation that the application programmer has carefully and purposely avoided at application level. It is possible to put on the programmer the extra burden of making sure that the progress engine never gets in the situation where it is performing some optimizations while `MPI_WIN_LOCK_ALL` and `MPI_WIN_ILOCK_ALL` are potentially pending; but we foresee no compensation for putting such a burden on the programmer.

Furthermore, applying the optimizations to fence epochs could either violate some MPI-3.0 constraints or create too much room for hazardous situations for the HPC programmer.

First, two adjacent fence epochs must be progressed serially because a middle fence entails a barrier semantics. In fact, epoch-closing fences always entail a barrier semantics. As for two adjacent epochs of which only the second one is fence-based, there is the increased difficulty of reasoning about disjoint memory accesses because each process of a fence epoch is potentially simultaneously an origin and a target. There is possibly some room for designing a simultaneous progression in this last case; but the complexity and required caution call for further careful analysis.

In summary, the optimization flags lead to the following additional rules:

12. If `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER` is set, then a GATS epoch created with `MPI_WIN_START` or `MPI_WIN_ISTART` is immediately activated if all the following conditions are simultaneously true:

    (a) The immediately preceding epoch is already activated. As per Rule 8, this condition actually means that all the preceding epochs are already activated.

    (b) The immediately preceding epoch was created by either `MPI_WIN_START`, `MPI_WIN_ISTART`, `MPI_WIN_LOCK` or `MPI_WIN_ILOCK`.

13. If `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER` is set, then an epoch created with `MPI_WIN_LOCK` or `MPI_WIN_ILOCK` is immediately activated if all the following conditions are simultaneously true:

    (a) The rank intended to be locked is not currently a target. This condition is required to 1) avoid violating the constraint that forbids to an RMA window to be simultaneously locked and exposed; and 2) prevent unintended recursive shared RMA lock acquisition by the progress engine.

    (b) The immediately preceding epoch is already activated.

    (c) The immediately preceding epoch was created by either `MPI_WIN_START`, `MPI_WIN_ISTART`, `MPI_WIN_LOCK` or `MPI_WIN_ILOCK`.

14. If `MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER` is set, then a GATS epoch created with

MPI_WIN_START or MPI_WIN_ISTART is immediately activated if all the following conditions are simultaneously true:

   (a) The immediately preceding epoch is already activated.
   (b) The immediately preceding epoch was created by either MPI_WIN_POST or MPI_WIN_IPOST.

15. If MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER is set, then an epoch created with MPI_WIN_LOCK or MPI_WIN_ILOCK is immediately activated if all the following conditions are simultaneously true:

    (a) The rank intended to be locked is not currently a target.
    (b) The immediately preceding epoch is already activated.
    (c) The immediately preceding epoch was created by either MPI_WIN_POST or MPI_WIN_IPOST.

16. If MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER is set, then an epoch created with MPI_WIN_POST or MPI_WIN_IPOST is immediately activated if all the following conditions are simultaneously true:

    (a) The local RMA window is not currently locked.
    (b) The immediately preceding epoch is already activated.
    (c) The immediately preceding epoch was created by either MPI_WIN_POST or MPI_WIN_IPOST.

17. If MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER is set, then an epoch created with MPI_WIN_POST or MPI_WIN_IPOST is immediately activated if all the following conditions are simultaneously true:

    (a) The local RMA window is not currently locked.
    (b) The immediately preceding epoch is already activated.
    (c) The immediately preceding epoch was created by either MPI_WIN_START, MPI_WIN_ISTART, MPI_WIN_LOCK or MPI_WIN_ILOCK.

18. For any two adjacent epochs, if at least one is `MPI_WIN_FENCE` or `MPI_WIN_IFENCE`, the second is not activated until the first one completes.

19. For any two adjacent epochs, if at least one is `MPI_WIN_LOCK_ALL` or `MPI_WIN_ILOCK_ALL`, the second is not activated until the first one completes.

We re-emphasize that the guarantee of hazard avoidance when the optimization flags are used is the responsibility of the programmer. It is the same responsibility that he bears when an `MPI_IRECV` or `MPI_ISEND` is issued while a previous `MPI_IRECV` or `MPI_ISEND` is still pending. If the programmer does not separate both calls by a barrier or a ***wait*** call, then it is assumed that he knows for sure that both nonblocking calls do not overlap in the memory regions that they modify. Similarly, if the programmer enables a flag for the progress engine to activate an epoch while a previous one is still active, then he is supposed to have the certainty that both epochs do not modify overlapping memory regions at either end of their communications.

5.7 Design and Realization Notes

As a proof of concept of the nonblocking synchronizations and epochs, we realize a concrete design that we implement in MVAPICH. The initial design strategies of RMA in the existing MVAPICH library were not adequate to use as a starting point. Consequently, instead of altering the MVAPICH RMA implementation, we went with a wholesale redesign. The new design does not simply cover nonblocking synchronizations; it covers the one-sided communication model as a whole and includes blocking synchronizations as well. The RMA communications have to be redesigned too. This section documents the key decisions and choices made for the design. The proof-of-concept is implemented over InfiniBand.

5.7.1 Separate and New Progress Engine

Nonblocking synchronization calls add a layer of complexity to the progress engine. In the existing MVAPICH design, RMA communication progression is too coupled with the epoch-closing synchronizations to allow the design alterations required by our proposal. In particular, while a closing synchronization call must still mean that no more RMA communication is allowed in a given epoch, there is now a need to decouple the exit of the synchronization routine from the completion of the RMA calls that came before. Such a requirement is a necessary condition for nonblocking epochs. Furthermore, the need to have multiple origin-side or target-side epochs simultaneously pending on a single RMA window is not compatible with any design that realizes the current MPI-3.0 RMA. Finally, new concepts are introduced that cannot be easily patched onto the existing means of handling MPI RMA inside MVAPICH.

We designate by *default progress engine* the progress engine that exists in the vanilla MVAPICH. MVAPICH has generic ways of transmitting data between processes. These generic ways define a uniform and higher-level abstraction to access the network device. Instead of using the existing network access abstraction, our new RMA progress engine has its own distinct and direct access to the network device. We found the existing middleware-level control message routing of MVAPICH heavyweight and potentially slow for our needs. As a result, the new progress engine puts in place its own lightweight and faster control message management. Examples of control messages in the case of one-sided communications are lock requests, lock release notifications, exposure granting notifications and intermediate operand buffer negotiation for accumulate operations that deal with large payloads. The distinct access to the network device does not duplicate any of the already created InfiniBand queues; both progress engines still use the same InfiniBand queues. However, if the default progress engine is the first to pick from the HCA a communication completion notification that belongs to the new progress engine, it forwards it; the new progress engine does the same if it discovers any InfiniBand completion element that belongs to the default progress engine. Finally, the new progress engine does use the already existing collective communication facilities internally if required. Such uses occur only at RMA window creation and freeing times. We explain later how the new one-sided communication design avoids resorting to collective communications even in fence epochs. It matters to emphasize that, with the exception of window creation and freeing time, the new progress engine never generates any MPI message queue item; this is done on purpose to avoid latencies. In comparison, because it internally uses generic two-sided operations for some of its communications, the default RMA design of MVAPICH generates message queue items.

The MPI middleware now has two distinct entry points into its progress engines. All one-sided communication-related calls delve into the new RMA progress engine. All the other MPI calls delve into the default progress engine. The routines shared by both one-sided and non one-sided communications delve into the default progress engine as well. Examples of such shared calls are the ***test*** and ***wait*** functions which uniformly complete two-sided and collective communications as well as the new nonblocking one-sided synchronizations. However, an MPI library conceptually has a unique centralized progress engine in order to allow generalized opportunistic message progression, among other things. For instance, when a nonblocking two-sided communication is pending, any MPI call that delves into the middleware for other unrelated purposes such as collective or RMA communications is expected to progress the pending two-sided communication if possible. As a result, our new RMA-only progress engine and the default one must behave from the point of view of the application as if they were the same. In order to realize that illusion, the one-sided communication progress engine always invoke the default progress engine as part of its own execution. Similarly, the default progress engine always invoke the new one-sided progress engine unless there is no RMA window object alive. Complex recursive invocations are controlled or prevented in various situations.

5.7.2 Deferred Epochs and Epoch Recording

Deferred epochs were already introduced in Section 5.6. An epoch is deferred when its immediate activation would violate a semantics rule. Deferred epochs are middleware-level concepts. A deferred epoch is *recorded* until it is closed at application level. Then, when it becomes activated later, it is *replayed* internally. Recording does not concern epochs that are activated at the same time as their application-level opening. We resort to two kinds of recordings, depending on how fast the subsequent replay can be fulfilled. The first type of recording, which is used for epoch opening routines, saves almost exactly the application-level arguments. The replay of such a recording reproduces almost the exact activity that would have occurred if the application-level call had been fulfilled in real time by the progress engine. The second kind of recording saves the application-level calls in various middleware-friendly forms. All the RMA communication calls, the epoch-closing functions as well as other passive target intra-epoch routines are saved with the second recording strategy.

An epoch can remain deferred even until it is closed at application level; in which case it is internally flagged as closed. Deferred epochs are hosted in a deferred epoch queue attached to their RMA windows. Every time an active epoch is completed internally for a given RMA window, the RMA progress engine scans all the existing deferred epochs of that RMA window and activates in sequence all those that can be activated. The scan stops when the first deferred epoch is encountered that fails the activation conditions.

5.7.3 Epoch Ids, Win traits and Epoch Matching

Accurate epoch pairing proved to be one of the most challenging aspects of the new design. With nonblocking synchronizations, processes are allowed to just issue an epoch as a whole and move on. Thus, epochs can pile up; and a certain peer can be far ahead of another one at application level. In particular, a target process could have issued its 100^{th} epoch before a given origin process, which is concerned by all these 100 exposure epochs, opens its first access epoch towards the same target; and vice versa. The problem is actually more complex than the linear gap between origins and targets in terms of epoch numbers.

For a more comprehensive statement of the problem, let us consider an RMA window W created over n processes. We designate by P_r the process of rank r in the group over which W is defined; and by T_i some set of processes all acting as targets and all belonging to the group over which W is created. The various sets of targets T_i are not necessarily disjoints. We consider GATS epochs as the explanation basis. Let us consider three processes P_0, P_1 and P_2. P_1 is defined to belong to T_0, T_1, T_2, T_3 and T_5. P_2 belongs to T_4 and T_5. P_0 is an origin that opens six epochs successively towards T_0 to T_5 in order. For P_0, knowing that the current access epoch is the 6^{th} is not very useful for epoch matching because all targets do not appear in all targets sets. Instead, one can notice that it is more important for P_0 to know how many accesses it has issued so far towards each distinct target. In particular, the 6^{th} access epoch of P_0 is the 5^{th} towards P_1 and the 2^{nd} towards P_2. The same reasoning can be made from targets to origins to show that exposure counters must be known for each individual origin; not for an exposure epoch as a whole. One can further notice that, due to the nonblocking synchronizations, P_2 could open its 2^{nd} exposure epoch far ahead of P_0 opening its overall 6^{th} access epoch to match the second exposure of P_2. Thus, there is a need to keep the history of all the granted accesses between any two peers so that the most recent granted access Id would not overwrite a previous one that is not yet consummed. In concrete terms, when a target grants access to an origin that is several epochs late, the granted access notification must persist for the origin to see it when it catches up.

The epoch Id matching calls for an unscalable design where a history of granted access from all possible targets must persist until each access notification is discovered and consumed. Notification queuing is reminiscent of message queuing; a situation that the design strives to avoid. We come up with an approach that keeps and manages the epoch notification history in $O(1)$ for both running time and space cost. Each RMA window over n processes has n `win_traits`. `win_traits` are numbered from 0 to $n-1$; and the `win_trait` bearing the number r describes the equivalent instance of the current RMA window in the process of rank r. If r happens to be the rank for the local process in the group spanned by the window, then `win_trait` r describes the local RMA window that hosts it. For conciseness, let us designate `win_trait` r by WT_r. `win_traits` do not exist for notification history management; they are required anyways for each RMA window in a given process to know a few information about the corresponding RMA window in the other processes encompassed by the window. For instance, WT_r contains the start address of the remote memory exposed by process r. Among other things, WT_r also allows the current process to know if it has locked the corresponding window in process r. In summary, WT_r is the object that facilitates for the local process a one-sided access to process r.

`win_traits` are created at window creation time. All the concerned processes exchange information via an internal All_gather call to update the `win_traits` of one another. An All_gather is a collective call that allows a group of processes to gather data from one another. One of the exchanged information is the virtual address of remote RMA windows. After the exchange, WT_r in the local process has the value of the pointer of the RMA window in process r. This address is not used in the local process; it is meant to be sent to the process r in notification packets if required. This approach allows the process r, when it receives a notification, to retrieve in a single operation the local RMA window the notification is meant for; avoiding the need for scanning queues of multiple RMA windows.

To solve the notification management issue, a `win_trait` WT_r in the local process P_l has an ordinal counter C_e that contains the Id of the last granted access received from process r. That number, which is a single 64-bit unsigned integer, is sufficient to hold the whole history of granted accesses from P_r to the local process P_l. C_e is directly updated by P_r without respect to how many accesses P_l has already requested. If P_r is on a remote node, the update occurs via RDMA. If P_r is in the same node, it occurs via inter-process shared-memory. When P_l requests the access number i towards P_r, if $i \leq C_e$, then P_l knows that the access is granted. $i > C_e$ means that P_r is yet to activate the exposure epoch that will grant the requested access. One can notice that this easy management of the notification history is favoured by the FIFO activation of epochs, as stated in Section 5.6.1. We emphasize that the FIFO activation specified in Section 5.6.1 was not meant to ease this aspect of the design; it was meant to avoid memory consistency hazards.

Passive target epochs do not have any target-side exposure epoch. However, the same epoch-matching mechanism is still required internally. An RMA window that grants a lock to a process P_l updates C_e inside P_l as if it was granting an access from an exposure epoch. No internal exposure epoch object is created in this case though.

When an origin-side epoch completes, it sends a `done` packet to each concerned target. A `done` packet must match the right exposure epoch as well. This is done trivially by sending back the ordinal Id number used by the completing origin-sided access. This number matches the ordinal Id of the target-side exposure that granted the access. For lock releases, a different kind of `done` packet is sent; so the RMA window could update its lock state.

5.7.4 Notes on Lock and Fence Epochs

Explicit RMA lock requests and lock releases are sent to the process hosting the window to lock or unlock. While the whole passive target synchronization involves both peers internally, the lock granting does occur one-sidedly. This approach is uniformly used for both `MPI_WIN_LOCK`, `MPI_WIN_LOCK_ALL` and their respective nonblocking equivalents.

We mentioned in Section 2.1.4 that a single fence call can close a previous fence epoch if any; and immediately open a new one. Such a fence was termed middle fence. If a middle fence is not intended, MPI defines `MPI_MODE_NOPRECEDE` as an assertion that can be passed to the fence call to hint that the call is not closing any previous epoch; it is only opening a new one. Similarly, `MPI_MODE_NOSUCCEED` is provided to mean that the current fence call closes the previous fence epoch but does not open a new one. The new progress engine never deals with middle fences directly. Instead, it internally breaks each middle fence in two separate fence calls; the first one bearing `MPI_MODE_NOSUCCEED` to only close the previous epoch; and the second one bearing `MPI_MODE_NOPRECEDE` to solely open a new fence epoch. We designate the closing and the newly opening epochs by F_0 and F_1 respectively for the sake of naming. A middle fence entails a barrier between the two epochs F_0 and F_1 that it separates. Our design bypasses any explicit barrier call. First of all, for any given participating process, the predicate that fulfills Rule 12 to Rule 19 in the design forbids the activation of F_1 until F_0 completes. The rules were stated in Section 5.6.2. For a given participating process, F_0 completes and terminates normally when it has completed all its RMA towards all the involved peers and it has received a `done` packet from all the other peers. F_0 can complete at different times on different participating processes. If F_0 has already completed on process P_i while it is still active on a subset S of processes that does not include P_i, then all processes in S are still not free to grant access to or to request access from P_i. Consequently, there is no hazard in activating F_1 right away on P_i; and this can be done without any barrier. There is no risk for any RMA communication originating from F_1 in P_i to reach any process of S and there is no risk for any process of S to try to reach F_1 in P_i.

5.8 Experimental Evaluation

We show comparison tests between the one-sided communication model provided by MVAPICH 2-1.9 and the new proof-of-concept implementation presented in Section 5.7. We term "MVAPICH", "New" and "New nonblocking" the three test series whose results are respectively obtained with the vanilla MVAPICH RMA, the new design with blocking synchronization and the new design with nonblocking synchronization. The experimental setup is an 8-node cluster. Each node has two Nehalem 2.6 GHz Pentium Xeon CPUs with hyperthreading disabled, 36 GB of memory and Mellanox ConnectX QDR InfiniBand HCA. The results of each microbenchmark test is the average over 100 iterations. For the application pattern and application tests, each result is the average obtained over 5 iterations.

5.8.1 Microbenchmark

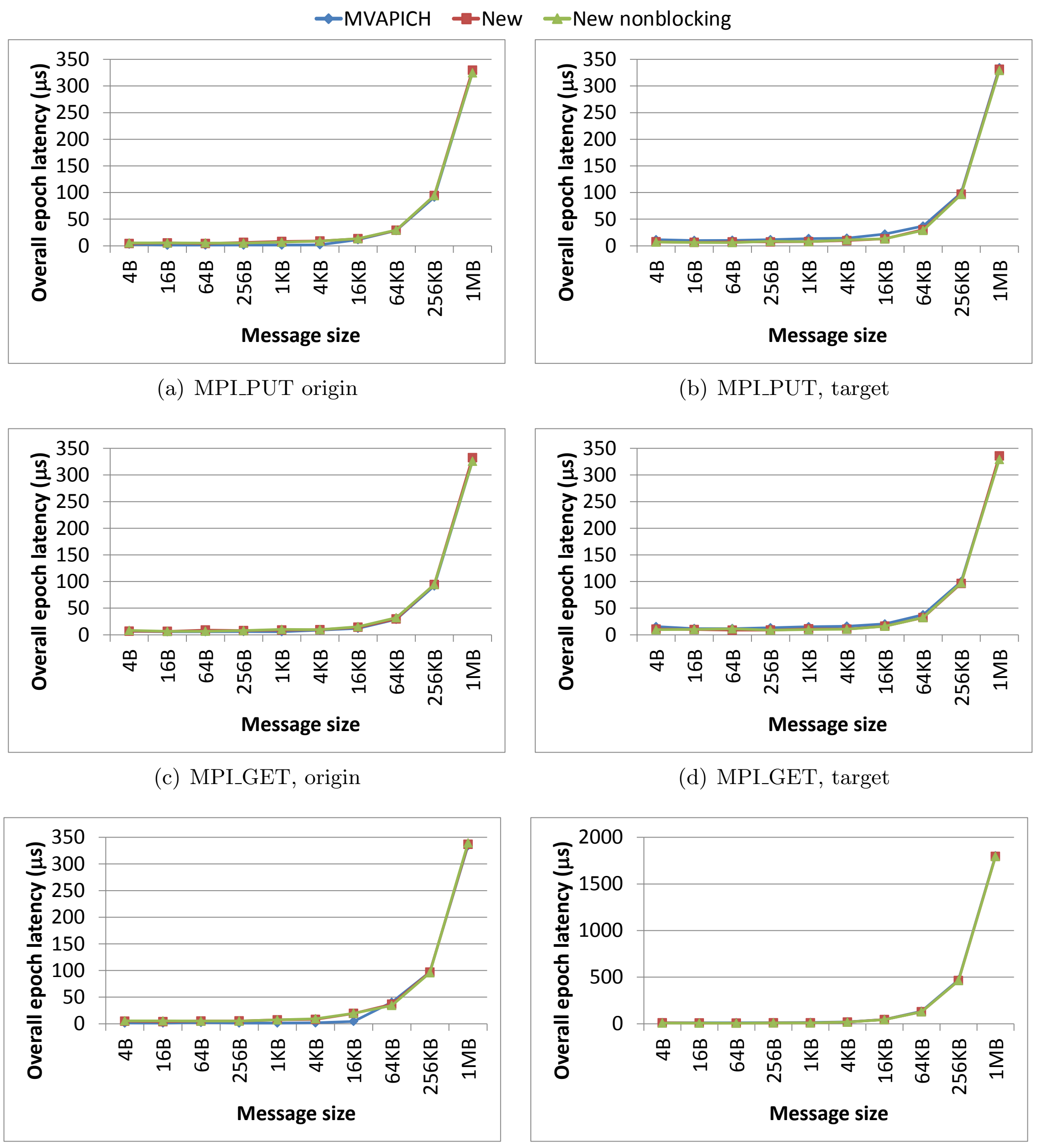
Generic Latency and Communication/computation overlapping

As discussed in Section 5.4, the claim of this chapter is the nonblocking synchronizations and epochs; as well as the aggressive epoch progression favoured by the info object-based flags. However, we deem important to first show that the realization of the aforementioned concepts does not come at the expense of some penalty or compromise on the performance of MPI one-sided communications in generic situations. Those generic situations occur when there is no scenario of inefficiency pattern and no situation where nonblocking synchronizations would trump what already exists in MPI-3.0 RMA.

The series of tests required to strictly compare two designs of MPI-3.0 RMA is large. As the research presented in this chapter is not about an alternative design of the existing MPI-3.0 RMA, we only present a few cases to show specific latency and communication-computation overlapping comparisons; and then we move on to concretely show the improvement brought by the nonblocking synchronizations and its accompanying new concepts.

In a series of tests meant for "generic latency" observations, we first compare GATS epochs (Figure 5.7), fence epochs (Figure 5.8) and lock epochs (Figure 5.9). Epochs based on `MPI_WIN_LOCK` represent the behaviours of both kinds of passive targets (`MPI_WIN_LOCK` and `MPI_WIN_LOCK_ALL`) for the sake of limiting redundant figures. For each kind of synchronization, tests are provided for epochs containing `MPI_PUT`, `MPI_GET` and `MPI_ACCUMULATE`. Each test is performed between one origin and one target. Even for fence epochs, only one process acts as origin and issues RMA calls; the other one behaves as a target. Finally, in order to cross-analyze communication latency and communication/computation overlapping in presence of RDMA, origins and targets are on remote nodes.

One can notice for both small and large messages that the latency behaviours are similar for all three test series. For all three synchronizations, one can also notice for target processes that `MPI_ACCUMULATE` is more expensive for messages of 16KB or more; and the observation uniformly applies to all three of MVAPICH, New and New nonblocking test series. For instance, epochs containing both `MPI_PUT` and `MPI_GET` takes about $100\mu s$ to complete for messages of 256KB. For messages of 1MB, the epochs complete in less than $350\mu s$. For `MPI_ACCUMULATE` in GATS and fence settings, while the origin-side epochs still lasts about $100\mu s$ and less than $350\mu s$ for 256KB and 1MB respectively (Figure 5.7(e), Figure 5.8(e)), the target-side epochs last $500\mu s$ and $1800\mu s$ respectively for 256KB and 1MB (Figure 5.7(f), Figure 5.8(f)). These increased target-side accumulate latencies are mostly due to a more noticeable computation overhead when the operand is bigger. In both MVAPICH and the new design, the target always does the computation of `MPI_ACCUMULATE`, especially when it is on a remote node. While the target must complete the computation before completing its epoch, the origin moves on as soon as its side of the operands is transferred to the target; explaining why the origin has a much lower latency. To a much lesser extent, buffer negotiation can add to the large-payload `MPI_ACCUMULATE` latency as well. In fact, for small-payload accumulates, the origin eagerly sends its side of the operands. The eagerly sent operand is received in generic pre-posted target-side buffers meant for eager communications. However, as already mentioned for the Rendezvous protocol in Section 2.1.2, system buffer is a scarce resource and cannot be pre-allocated in large chunks. As a result, intermediate buffers for large accumulate operands must be allocated on-demand. There is therefore a back-and-forth of control messages between the origin and the target for the target to create a temporary intermediate operand buffer whose remote access information are sent to the origin. This Rendezvous-like activity is the buffer negotiation.



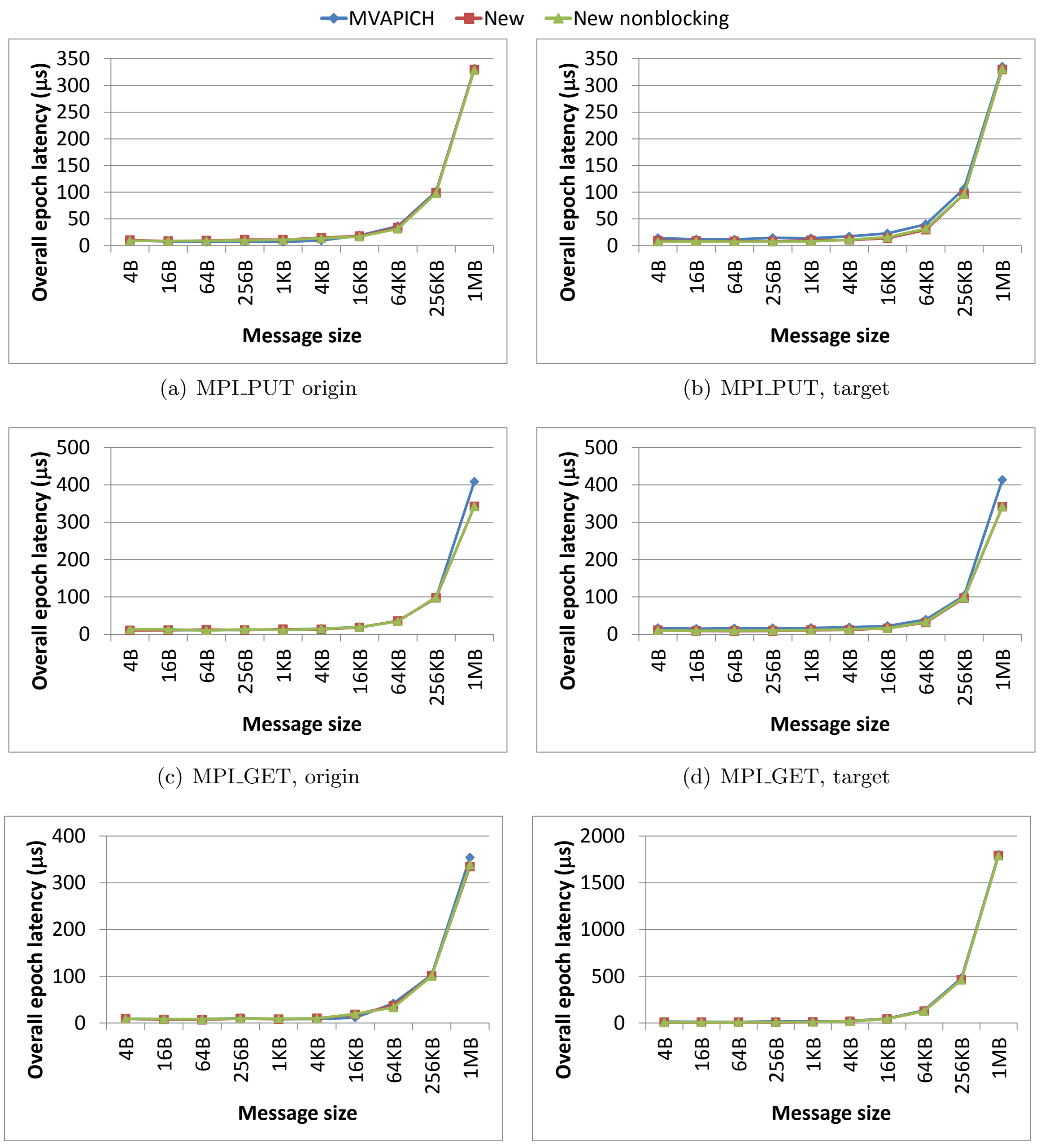
(a) MPI_PUT origin (b) MPI_PUT, target

(c) MPI_GET, origin (d) MPI_GET, target

(e) MPI_ACCUMULATE, origin (f) MPI_ACCUMULATE, target

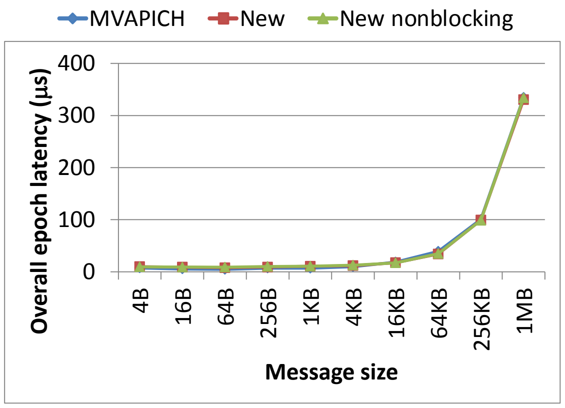
Figure 5.7: Comparative overall GATS epoch latency of MVAPICH, blocking and nonblocking versions of the new design

For passive target (Figure 5.9(c)), the increase in `MPI_ACCUMULATE` latency for large operands is noticed at origin-side. As per the MPI specification, the origin-side completion of passive target epochs cannot occur until the associated RMA communication is guaranteed to have

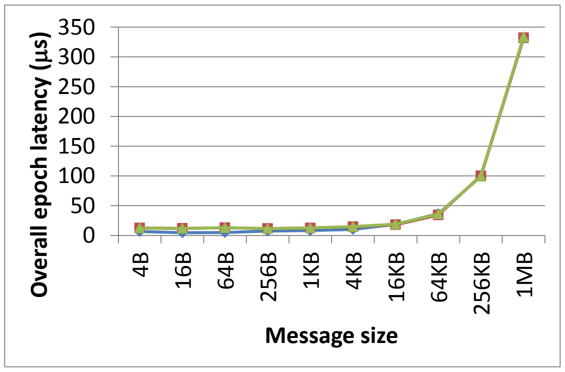


(a) MPI_PUT origin
(b) MPI_PUT, target
(c) MPI_GET, origin
(d) MPI_GET, target
(e) MPI_ACCUMULATE, origin
(f) MPI_ACCUMULATE, target

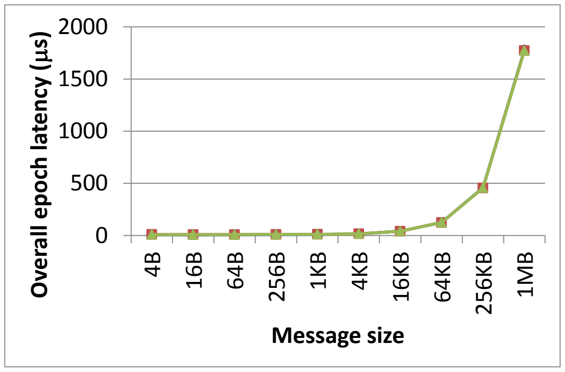
Figure 5.8: Comparative overall fence epoch latency of MVAPICH, blocking and nonblocking versions of the new design



(a) MPI_PUT origin



(b) MPI_GET, origin



(c) MPI_ACCUMULATE, origin

Figure 5.9: Comparative overall lock epoch latency of MVAPICH, blocking and nonblocking versions of the new design

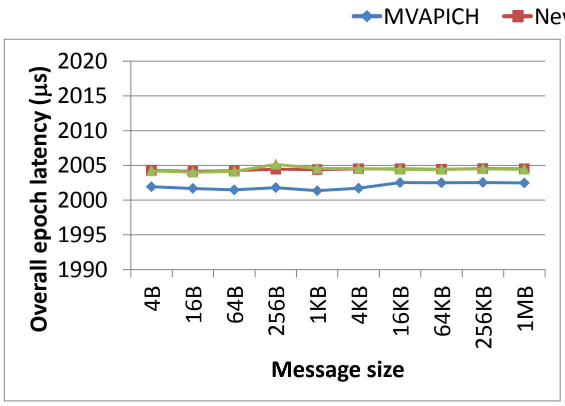
completed, not just at the origin-side as it is the case for the other synchronizations; but at the target-side as well. The observance of that rule leads to the origin incurring the target-side latency of `MPI_ACCUMULATE` latency as well. In practice, for a passive target epoch E_i, neither the vanilla MVAPICH design, not our new design delays the origin until the operation actually completes on the target-side. This is an optimization meant to allow the origin to move on earlier. However, if the origin must open a subsequent epoch E_{i+1}, the activation of E_{i+1} must first check that the previous E_i had already completed on the target. Due to the test being averaged over multiple iterations, the effect of the origin moving on hastily for a given epoch is paid for in the subsequent epoch; and the actual target latency ends up being felt in the origin-side measurements.

We assess communication/computation overlapping by using the same methods as in Section 3.3. An activity made of an RMA communication and a computation is hosted inside an epoch. The computation time c_p is chosen to be big enough to entirely cover all the communication data transfer latency for all the chosen message sizes. c_p is chosen to be $2000\mu s$, so as to account for the high latency of large-operand accumulates. Communication/computation overlapping occurs if, as per Equation 3.3, the overall latency d of the activity is independent of the message size and is about the length of the computation. In absence of communication/-computation overlapping (Equation 3.1), the overall latency of the activity increases with the message size. In Chapter 3, the fixed portion ε of communication duration was created by two-sided calls. In this case, it is created by the synchronization calls, the equivalent of the RMA call with a zero-byte payload, and the ***wait*** call in the case of nonblocking synchronizations.

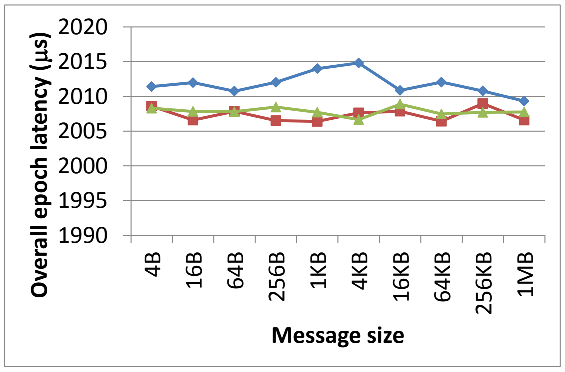
For GATS (Figure 5.10) and fence (Figure 5.11), the computation of length c_p is inserted in both the origin-side and the target-side epochs; rendering the CPU unavailable on both sides. For `MPI_PUT` and `MPI_GET` in the cases of both GATS (Figure 5.10(a), Figure 5.10(b), Figure 5.10(c), Figure 5.10(d)) and fence (Figure 5.11(a), Figure 5.11(b), Figure 5.11(c), Figure 5.11(d)), one can observe that the curves oscillate between $2000\mu s$ and $2020\mu s$, that is, c_p plus some small increment that represents ε. Communication/computation overlapping is therefore achieved for `MPI_PUT` and `MPI_GET` in GATS and fence settings for all three test series. For `MPI_ACCUMULATE`, communication/computation overlapping is not achieved from 16KB onwards (Figure 5.10(e), Figure 5.10(f), Figure 5.11(e), Figure 5.11(f)); and this observation is the result of intermediate operand buffer negotiation. The back-and-forth of control message required by the buffer negotiation requires CPU intervention for the detection of each incoming control message and the triggering of the transfer of each outgoing control message. The conclusion for GATS and fence is that the new design achieves communication/computation overlapping just as the existing MPI-3.0 RMA in MVAPICH.

For passive target, while the same buffer negotiation issue prevents communication/computation overlapping for large-operand accumulates for all three test series (Figure 5.12(c)), one can notice that only the two flavours of the new design realize communication/computation overlapping for `MPI_PUT` and `MPI_GET`. MVAPICH-RMA simply does not achieve any communication/computation overlapping for passive target because it uses a lazy lock acquisition scheme; an issue for which a workaround was proposed for MPICH in [116]. In Figure 5.12(a) and Figure 5.12(b) no communication/computation overlapping is achieved even for message sizes below 16KB; the figures do not show the lack of overlapping of MVAPICH for 4KB and below because of the low communication latencies of these small message sizes. In lazy lock acquisition, the origin does not internally issue the lock request when the application calls `MPI_WIN_LOCK`. As a result, any RMA call issued inside the epoch is always locally queued and deferred even if the lock is available on the target. Then, when the application issues `MPI_WIN_UNLOCK`, the actual lock request occurs and the RMA communications are fulfilled. The lazy lock acquisition internally degenerates the whole epoch to `MPI_WIN_UNLOCK`, voiding any chance of overlapping any computation hosted inside the epoch with any communication issued inside the same epoch.

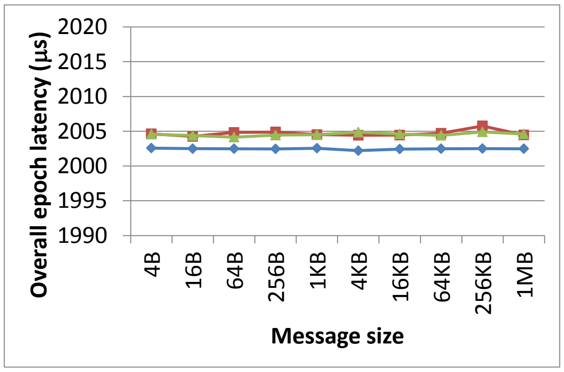
We mentioned for Figure 5.9(c) that origin processes incur the target-side latency of `MPI_ACCUMULATE` computation because of the need for origins to wait for target-side completion in passive target epochs. However, in Figure 5.12(c) the overall epoch latencies of messages of size 16KB, 64KB, 256KB and 1MB do not seem to reach the values observed in Figure 5.10(f) and Figure 5.11(f). In fact, as previously discussed for Figure 5.9(c) in the "generic latency" tests, the target-side completion of epoch E_i is still checked by a subsequent epoch E_{i+1}. However, that extra target-side completion latency associated with epoch E_i is overlapped by the work inserted in E_{i+1} in the case of MVAPICH. In the case of the new design, the extra latency of the target-side completion is overlapped with the work inserted inside E_i.



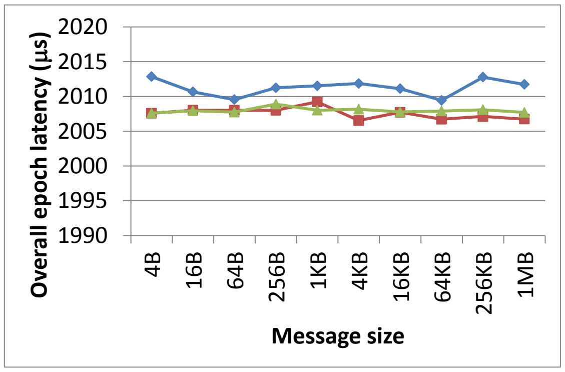
(a) MPI_PUT origin



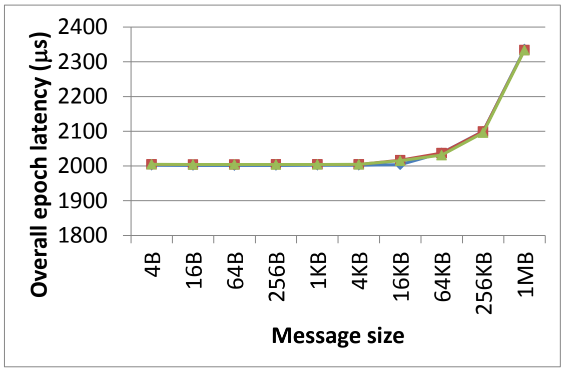
(b) MPI_PUT, target



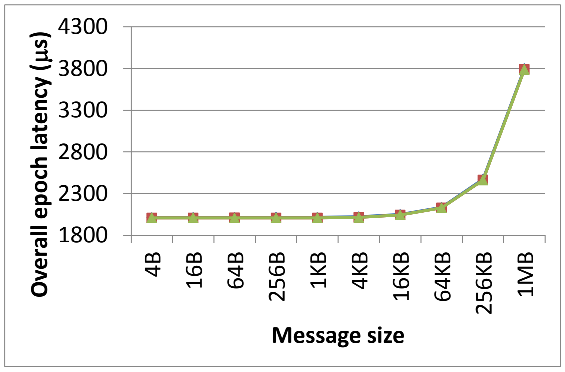
(c) MPI_GET, origin



(d) MPI_GET, target

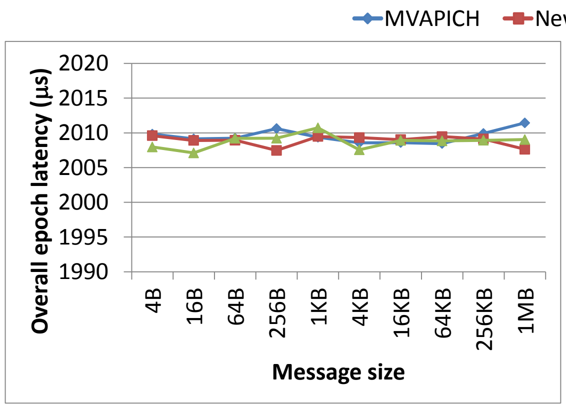


(e) MPI_ACCUMULATE, origin

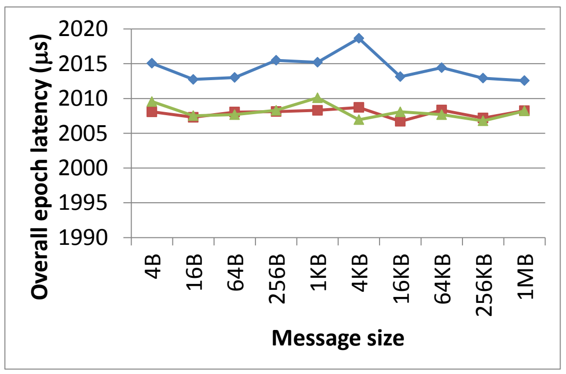


(f) MPI_ACCUMULATE, target

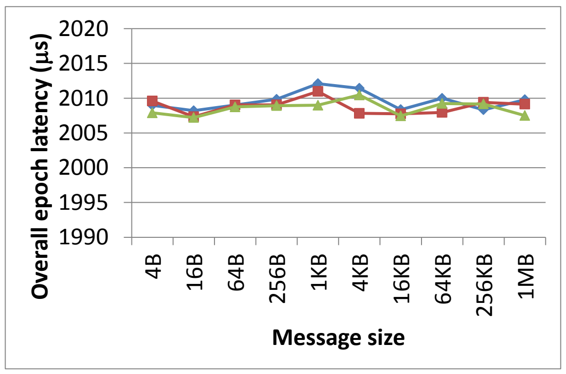
Figure 5.10: GATS comparative communication/computation overlapping of MVAPICH, blocking and nonblocking versions of the new design



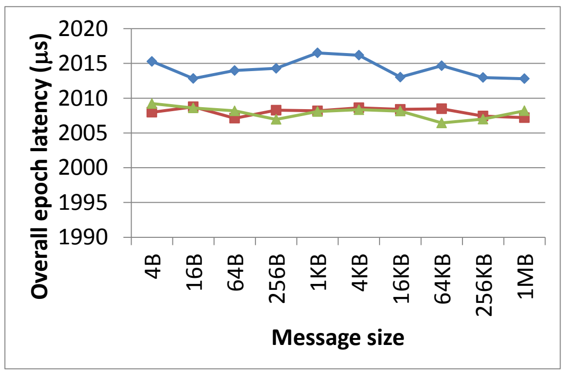
(a) MPI_PUT origin



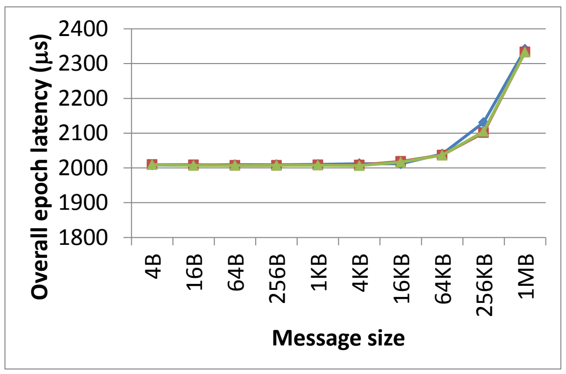
(b) MPI_PUT, target



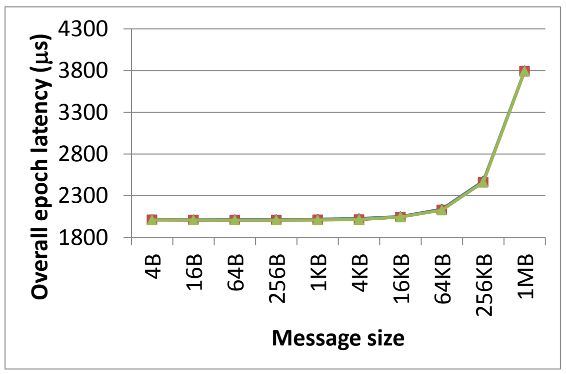
(c) MPI_GET, origin



(d) MPI_GET, target

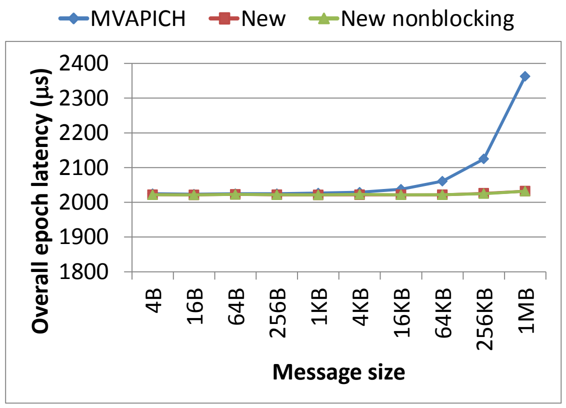


(e) MPI_ACCUMULATE, origin

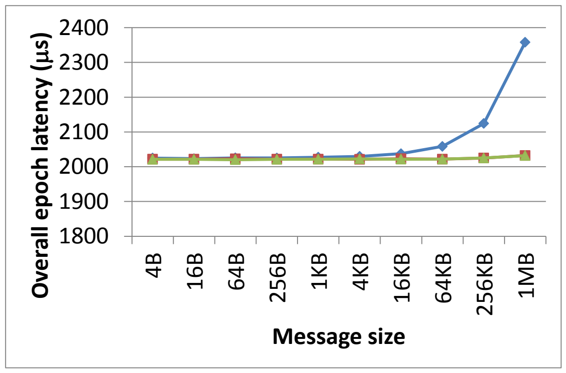


(f) MPI_ACCUMULATE, target

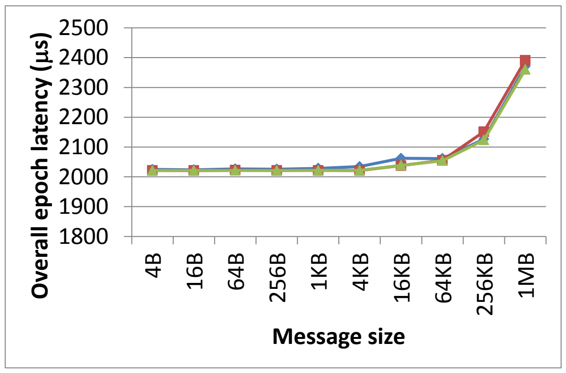
Figure 5.11: Fence comparative communication/computation overlapping of MVAPICH, blocking and nonblocking versions of the new design



(a) MPI_PUT origin



(b) MPI_GET, origin



(c) MPI_ACCUMULATE, origin

Figure 5.12: Passive target comparative communication/computation overlapping of MVAPICH, blocking and nonblocking versions of the new design

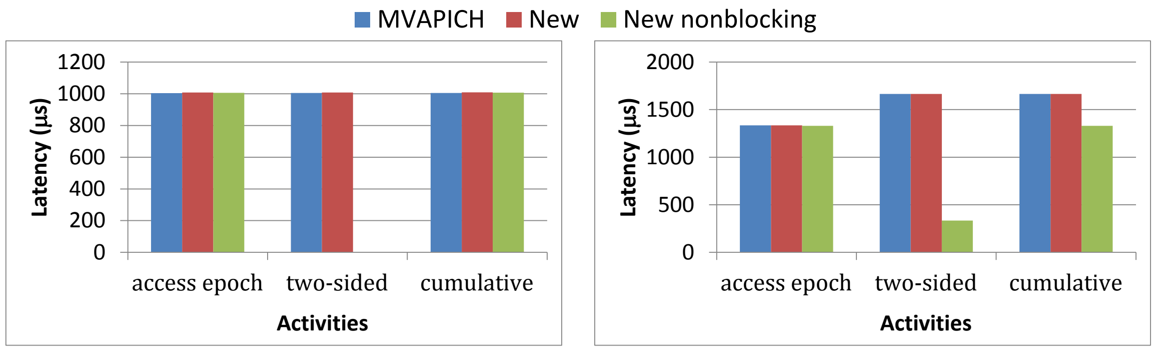
The Inefficiency Patterns

Late Post: The test setting is made of an origin process, a target process and a third process meant for two-sided communications. The target process is 1000μs late in opening its exposure epoch; leading to the Late Post situation. The origin first opens an access epoch towards the target, completes the epoch and then performs a subsequent activity. The subsequent activity is chosen to be a nonblocking two-sided communication towards the third process; however, it could have been anything else, including another epoch. The access epoch hosts a single `MPI_PUT` just for the sake of doing some RMA communication. Both the `MPI_PUT` of the first activity and the two-sided communication of the subsequent activity ship 1B of data for the small message case and 1MB of data for the large payload case.

We show in Figure 5.13 how long the access epoch takes to complete, how long the subsequent activity takes to complete and how long the origin takes to complete both activities. The time origin for all three measurements is taken at 0.

The delay of the late post cannot be avoided by the origin-side epoch trying to access the late exposure. As shown in Figure 5.13, the access epoch length is at least $1000\mu s$ for all three test series. However, while the blocking synchronizations propagate that delay to the subsequent activity as well, the nonblocking synchronization does not allow it to go beyond the sole access epoch that it directly impacts. As proof, one can notice that the two-sided activity only takes about $1.74\mu s$ for the 1B case (Figure 5.13(a)); and $334.32\mu s$ for the 1MB case (Figure 5.13(b)) with nonblocking synchronizations. In comparison, it takes more than $1000\mu s$ (Figure 5.13(a)) and more than $1660\mu s$ (Figure 5.13(b)) respectively for the 1B and 1MB cases for the two test series that use blocking synchronizations. The large payload case (Figure 5.13(b)) is particularly interesting because it shows that the two-sided activity is delayed by both the Late Post inefficiency and the data transfer latency of the access epoch when blocking synchronizations are used. In the nonblocking synchronization case, the activities get internally reordered in the progress engine; and the two-sided activity ends up incurring only the latency of its own data transfer. In fact, the subsequent activity gets overlapped with the delay of the Late Post inefficiency. Consequently, the cumulative latency of both activities in the case of nonblocking synchronizations end up simply being the overall latency of the access epoch only; as if the second activity occurred for free and without any latency at all. It has to be mentioned that if the second activity bears any data dependency with the first one, then the programmer has to ensure that it occurs after the epoch completes. In that case, the use of blocking synchronizations becomes mandatory. Alternatively, the second activity can be set to start only after the exit of the wait call associated with the first activity if nonblocking synchronizations must be used.

It is possible to show with more than a single subsequent activity that blocking synchronizations could endlessly propagate a single Late Post inefficiency delay from one subsequent activity to another one throughout the entire lifetime of an HPC job. In comparison, nonblocking synchronizations offer means of absorbing these delays from one subsequent activity to another one until they disappear entirely.



(a) Small data (1B)

(b) Large data (1MB)

Figure 5.13: Mitigating the Late Post inefficiency pattern: observing delay propagation in an origin process

Late Complete: The test setting is made of a single origin and a single target. The origin issues a single `MPI_PUT` and then, in order to attempt communication/computation overlapping, it works for 1000μs before the blocking call that completes the epoch. As a reminder, epoch completion and epoch closing all correspond to the closing synchronization in the blocking case. In the nonblocking synchronization case, epoch completion is separate from epoch closing and is fulfilled by invoking one of the ***wait*** or ***test*** family of routines. This test is performed for multiple message sizes. Since even the largest payload, which is 1MB, takes less than 350μs to transfer, a work that lasts 1000μs delays the epoch beyond the actual length of the RMA transfer; leading to the Late Complete inefficiency. Figure 5.14 shows the impact of that delay on the target, which is the victim. One can notice that none of the delay is propagated to the target when nonblocking synchronizations are used. A payload of 4B takes 9.05μs while a payload of 1MB takes 336.1μs as if the origin performed no delaying work. In comparison, the two test series using blocking synchronizations transfer the totality of the delay to the target. In this case, due to communication/computation overlapping, the target must only wait for the 1000μs of origin-side computation time, which is the bigger of the two latencies of communication or computation.

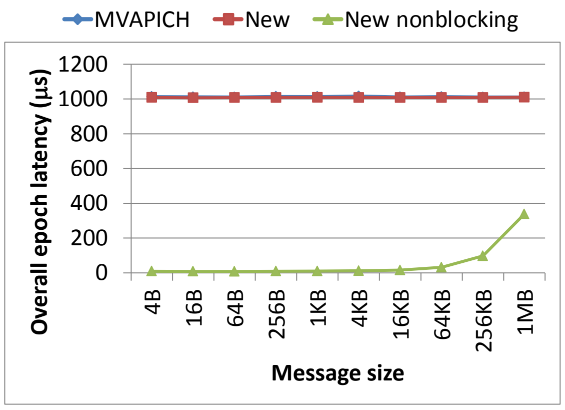


Figure 5.14: Mitigating the Late Complete inefficiency pattern: observing delay propagation in a target process

Early Fence: The test setting is made of two processes and a fence epoch. One of the processes acts like the origin and issues the RMA communications while the other one acts like a pure target and does not issue any communication. The origin process issues a single `MPI_PUT` of either 256KB or 1MB. The message sizes must all be large in this case because the Early Fence inefficiency occurs when the target issues an epoch-closing fence while the RMA transfers are still in progress. As a result, the test requires the transfers to last.

Unlike the Late Post inefficiency, the Early Fence inefficiency is not the result of any delay. In fact, an Early Fence inefficiency can occur even when all the involved processes are on time and no peer creates any non RMA-related delay. However, since the wait created by an early epoch-closing fence call corresponds to an idling CPU core, the Early Fence situation is still inefficient from an HPC point of view and should therefore be mitigated.

The early fence situation cannot be avoided by the immediate epoch that it impacts. Thus, the wait mitigation can be achieved by occupying the CPU with some subsequent activity. The observation in this test can be similarly done from either the origin side or the target side as they are both impacted by the Early Fence inefficiency exactly in the same way. We simply choose the target as the side where the transfer latency mitigation is to occur. Thus, the target does some CPU-bound work right after the epoch. We use a synthetic work of $1000\mu s$. As shown in Figure 5.15, the overall latency of the fence and the subsequent activity is the sum of both activities with the blocking synchronization tests. However, with the nonblocking synchronization test, the wait of the early fence is completely mitigated by the subsequent activity; leading to the target only incurring barely more than the $1000\mu s$ of the subsequent activity for both activities. One should notice that the Y axis in Figure 5.15 starts at $1000\mu s$ so as to emphasize the overall duration of both activities in the case of nonblocking synchronizations. We show in this test that nonblocking synchronizations allow any RMA transfer that is pending after an early fence call to be overlapped with CPU-bound activities performed after the epoch.

It helps to remind that the same wait mitigation performed by the nonblocking synchronization cannot be achieved by inserting the work before the epoch-closing fence, so as to allow even the blocking synchronizations to achieve communication/computation overlapping. In fact, inserting the work before the epoch-closing fence creates the risk of the Wait at Fence inefficiency; and is not a risk-free way of mitigating the Early Fence situation. Inserting the work after the fence does not bear any risk of leading to another inefficiency pattern.

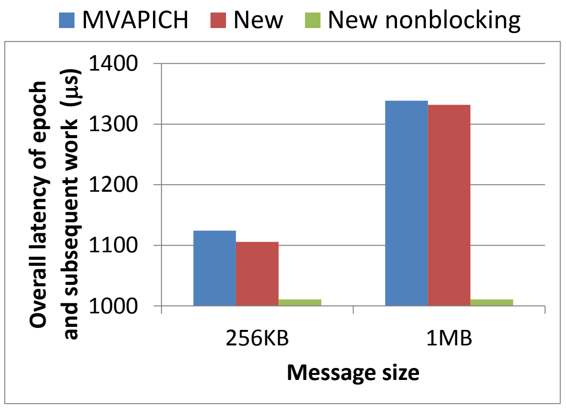


Figure 5.15: Mitigating the Early Fence inefficiency pattern: observing communication latency propagation in a target process

Wait at Fence: The Wait at Fence inefficiency pattern can be seen as the fence epoch equivalent of the Late Complete inefficiency pattern for GATS epochs. The test setting for Wait at Fence is therefore the same one used for Figure 5.14. We resort to one origin, one target; with the origin working for longer than required before completing its epoch. The work length is $1000\mu s$. The opening and closing fences respectively use `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED`. As expected, one can notice on Figure 5.16 that nonblocking synchronizations allow the target to avoid incurring any of the non RMA-related delay created by the origin-side work. The situation is different for blocking synchronization tests where the target must wait for the bigger of the data transfer latency or origin-side work duration.

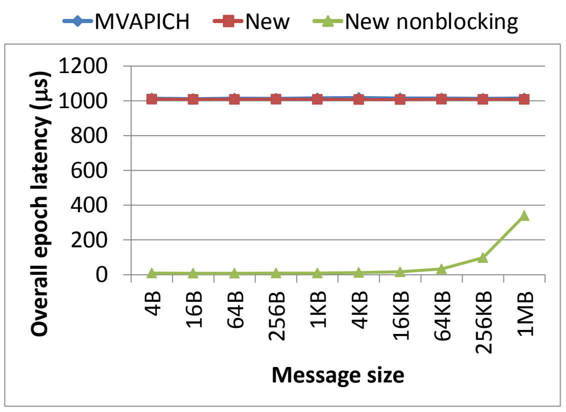
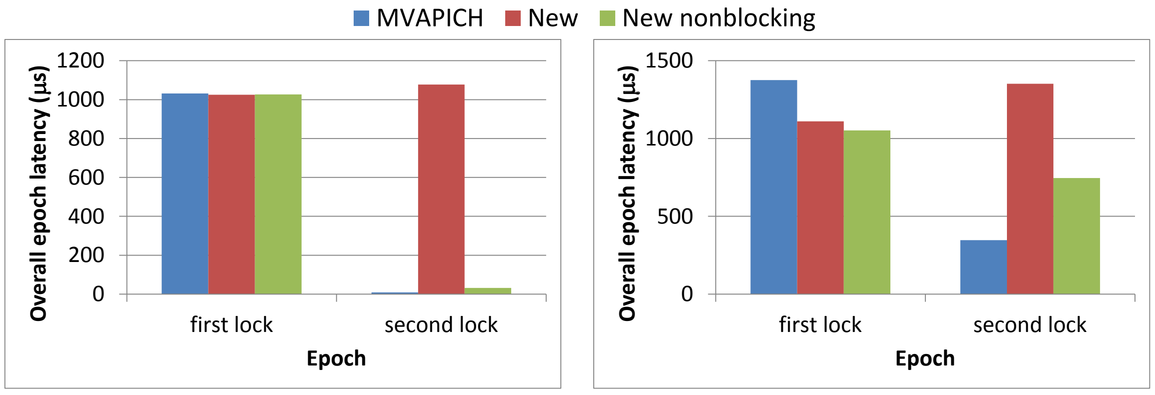


Figure 5.16: Mitigating the Wait at Fence inefficiency pattern: observing delay propagation in a target process

Late Unlock: This test requires two origin processes O_0 and O_1 and a single passive target. Both origins request the same RMA lock from the target; but we ensure that O_0 sends its lock request first. Then O_0 tries to overlap $1000\mu s$ of work with the data transfer of its epoch. O_1 does not perform any work inside its epoch. Each origin issues a single 1B or 1MB `MPI_PUT`. The results shown in Figure 5.17 present how long the first lock epoch, issued by O_0, took to complete; and how long the second lock epoch, issued by O_1, took to complete. We want to observe the second lock epoch to see if it is impacted by the late unlock of O_0.

One can first notice that MVAPICH does not suffer the Late Unlock issue. In fact, thanks to the lazy lock acquisition, O_0 does not actually send the lock request in `MPI_WIN_LOCK`. All lock requests are actually issued internally in `MPI_WIN_UNLOCK`. Thus, when O_0 gets plunged in the delaying work, the lock is still available for O_1 to get it without wait. However, as previously mentioned in Figure 5.12, lazy lock acquisition comes at the cost of a complete absence of communication/computation overlapping. On Figure 5.17(b), the absence of communication/-computation overlapping of MVAPICH is, once again, shown because the overall latency of the first epoch is $1375.61 \mu s$. In comparison, the two new design test series keep the overall latency of the first lock epoch a bit above $1000 \mu s$ thanks to communication/computation overlapping.



(a) Small data (1B) (b) Large data (1MB)

Figure 5.17: Mitigating the Late Unlock inefficiency pattern: observing delay propagation to a subsequent lock requester

In the new design, because the lock is effectively granted to O_0 before it gets plunged in a computation, the blocking synchronization inflicts a Late Unlock inefficiency to the second epoch. In Figure 5.17(a), the second epoch lasts more than $1000 \mu s$ to transfer just 1B of data. In Figure 5.17(b) as well, the latency of the second epoch is far beyond $1000 \mu s$, still due to the delay of the first epoch being transferred to the second one.

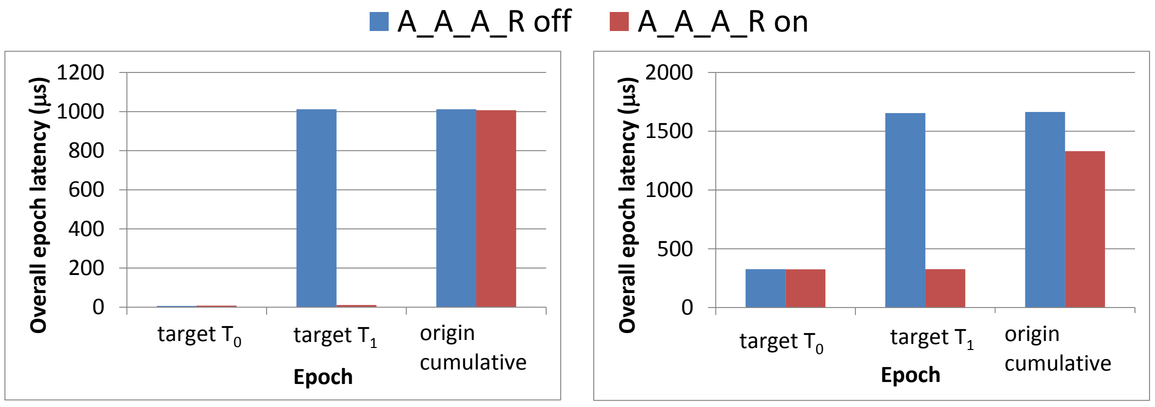
Finally, one can notice that the nonblocking synchronization fixes the Late Unlock issue on top of allowing communication/computation overlapping. In both Figure 5.17(a) and Figure 5.17(b), the second epoch lasts well below $1000 \mu s$; meaning that the work delay in the first epoch is not propagated to the second one. One can nevertheless notice that the second epoch in the case of the nonblocking synchronization still lasts longer than in the case of MVAPICH. This difference in latency is due to the second epoch waiting for the RMA communication of the first lock to complete. Thus, starting from time zero, the overall latency of the second epoch in the new nonblocking synchronization design is the sum of the duration of the first epoch, without the delaying work, and the duration of the second epoch from the point where it actually gets a hold of the lock.

The Progress Engine Optimization Flags

All the tests in this subsection are only based on nonblocking synchronizations. The results compare how fast the progress engine deal with two adjacent epochs, both issued in nonblocking ways; with and without some specific optimization enabled.

MPI_WIN_ACCESS_AFTER_ACCESS_REORDER: `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER` (`A_A_A_R`) is an origin-side optimization that concerns GATS and lock epochs. For the GATS test, we consider a single origin O and two targets T_0 and T_1. The origin opens an access epoch towards T_0 first and then towards T_1. In each epoch, a single `MPI_PUT` of 1B or 1MB is issued. The exposure of T_0 is $1000\mu s$ late; leading to a Late Post situation. The observations of interest are the latency of the epoch of T_1 as well as the overall latency of the epochs of O_1.

Without `A_A_A_R` enabled, the progress engine does not activate the second access epoch until the first one completes. The delay that is propagated to the second access epoch impacts the second target T_1 as well as the cumulative time required for the origin to complete both access epochs. The delay propagation is more visible with the 1MB payload Figure 5.18(b) where the second target epochs takes $1654.73\mu s$ to complete. Knowing that 1MB takes about $330\mu s$, the second target effectively incurs the latency of its own RMA data transfer as well the latency of the RMA transfer of the first target epoch and the $1000\mu s$ delay. The cumulative origin-side latency, which is $1663.69\mu s$ is impacted by the $1000\mu s$ delay as well. Enabling `A_A_A_R` allows the progress engine to activate the second origin-side access epoch and even complete it ahead of the first one. The delay is not propagated; and the second target only incurs its own epoch latency. The latency of the second epoch is overlapped with the delay of the first Late Post; leading to a smaller cumulative latency for the origin. This test scenario is similar to the one used for Late Post. In this case, we replace the subsequent activity by another epoch. Due to the default FIFO progression of epochs inside the progress engine, the



(a) Small data (1B) (b) Large data (1MB)

Figure 5.18: Out-of-order GATS access epoch progression with `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER`

subsequent activity must still wait inside the middleware in spite of the use of nonblocking synchronizations; unless `A_A_A_R` is enabled.

To test `A_A_A_R` with lock epochs, we consider two origins O_0, O_1 and two targets T_0, T_1. O_0 gets a lock from T_0 and works $1000\mu s$ before releasing the lock. Right after O_0 gets the lock, O_1 requests the same lock from T_0. Then O_1 request a subsequent lock from T_1. The observation of interest is the cumulative duration of both epochs of O_1. In each epoch, O_1 issues a single `MPI_PUT` of 1B or 1MB. Once again, the activation of `A_A_A_R` allows O_1 to complete its second epoch, the one towards T_1, out-of-order and not suffer the delay of its first access that must wait $1000\mu s$ to get a hold of the lock towards T_0 (Figure 5.19). As a consequence, the latency of the second epoch is not visible in the cumulative latency of both epochs because it overlaps the delay incurred by the first epoch. The Y axis in Figure 5.19 starts at $1000\mu s$ on purpose to emphasize the difference of latency with and without `A_A_A_R` activated. The difference of latency for the 1B cumulative results is not easily noticeable; but these results are nevertheless provided for the sake of completeness.

MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER: The test setting of `MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER` (`A_A_E_R`) is made of three processes P_0, P_1 and P_2. P_0 is an origin and P_1 is a target. P_2 behaves as a target for P_0 and then as an origin for P_1; in that order. P_0 is $1000\mu s$ late. Each access epoch issues a single `MPI_PUT` of 1B or 1MB. As

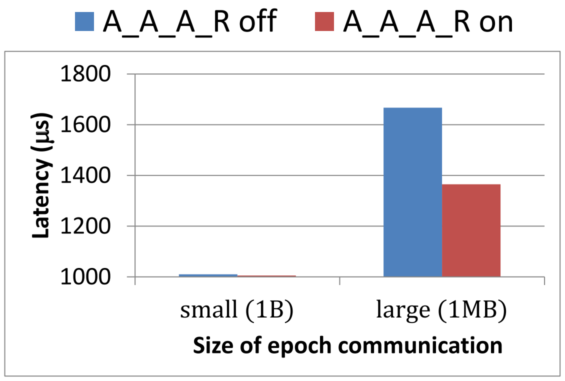
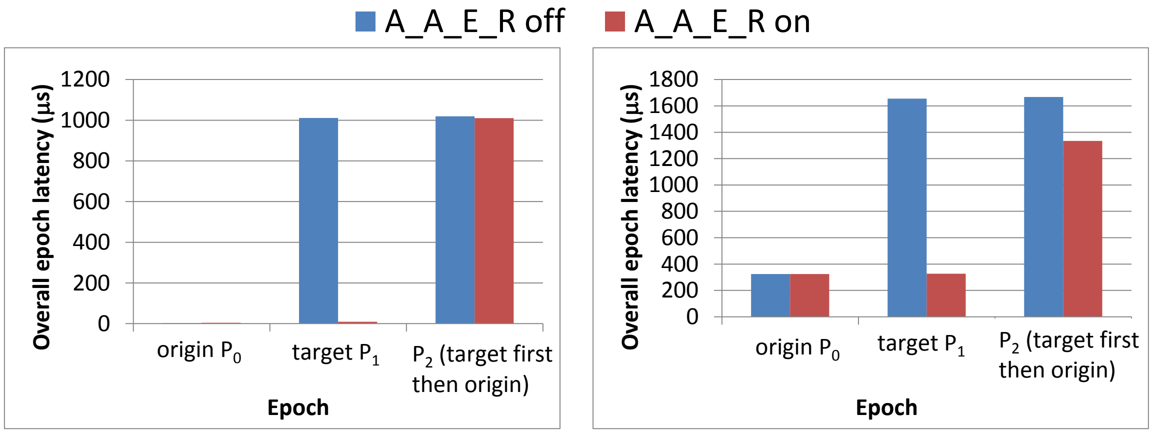


Figure 5.19: Out-of-order lock epoch progression with `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER`

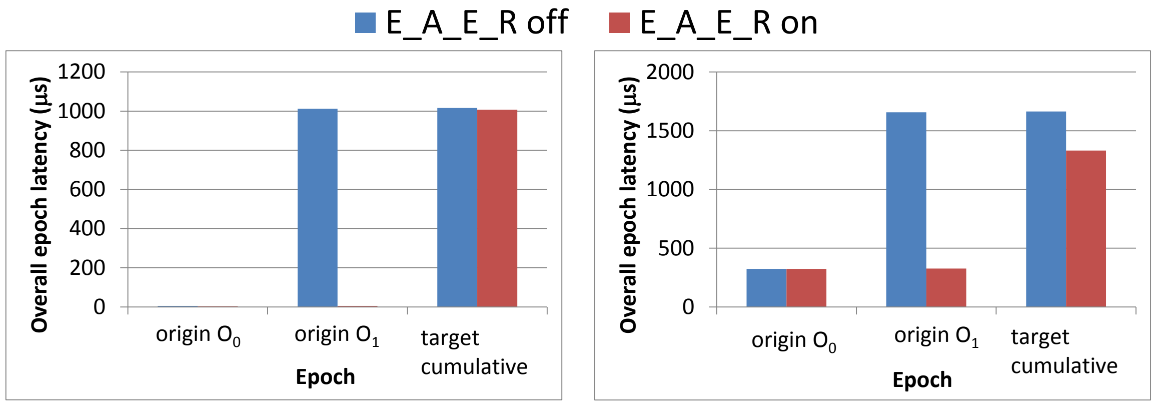


(a) Small data (1B) (b) Large data (1MB)

Figure 5.20: Out-of-order GATS epoch progression with `MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER`

shown on Figure 5.20, by default, the delay of P_0 is transferred to both P_1 and P_2. However, when P_2 enables `A_A_E_R`, its second epoch, meant for P_1 is progressed and completed before the first epoch; leading to P_1 not incurring the delay created by P_0. As a consequence of its second epoch overlapping the delay of the first epoch, the cumulative latency of P_2 is lowered as well.

MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER: The test setting of `MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER` (`E_A_E_R`) is made of two origins O_0, O_1 and a target. O_0 is $1000\mu s$ late. The first exposure of the target is meant for O_0 and the second for O_1. Each access epoch issues a single `MPI_PUT` of 1B or 1MB. Figure 5.21 shows that the delay



(a) Small data (1B) (b) Large data (1MB)

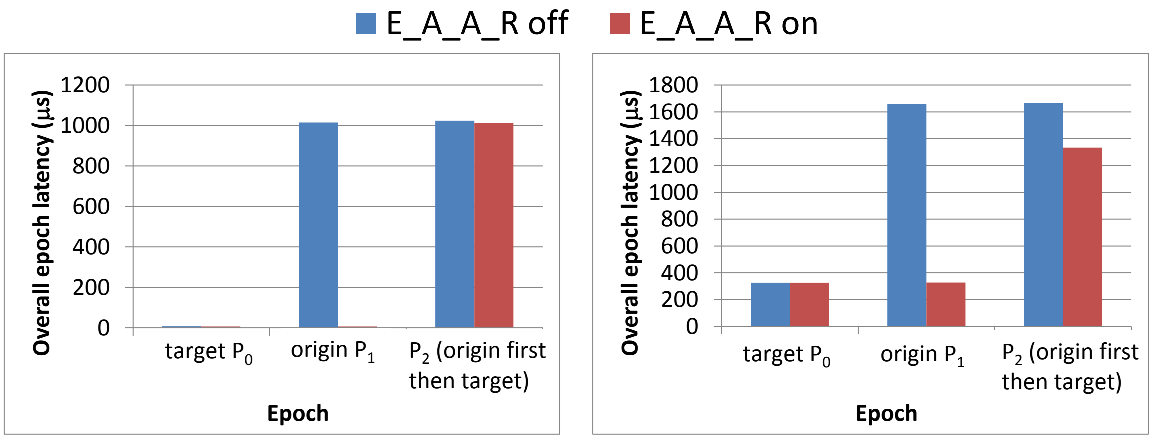
Figure 5.21: Out-of-order GATS epoch progression with `MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER`

of O_0 is transferred to O_1 by default; and the cumulative latency experienced by the target is the sum of that delay and the latency of both its epochs towards O_0 and O_1. However, when the target enables `E_A_E_R`, the delay propagation is prevented. Thanks to the second epoch overlapping with the delay, the cumulative latency experienced by the target is only made of the overall latency of the first epoch.

MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER: The test setting of `MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER` (`E_A_A_R`) is made of three processes P_0, P_1 and P_2. P_0 is a target and P_1 is an origin. P_2 behaves as an origin for P_0 and then as a target for P_1. P_0 is $1000\mu s$ late. Each access epoch issues a single `MPI_PUT` of 1B or 1MB. Once again, unlike the default case, the activation of `E_A_A_R` by P_2 prevents the propagation of the delay of P_0 to P_1 and allows the second epoch of P_2 to overlap the delay; leading to a lower cumulative latency for P_2 (Figure 5.22).

5.8.2 Unstructured Dynamic Application Pattern

We reproduce in this section the massively unstructured atomic communication pattern described in Section 5.4.2. The processes in the job host some distributed data that is the target of multiple atomic updates or transactions. In order to guarantee a large number of in-flight



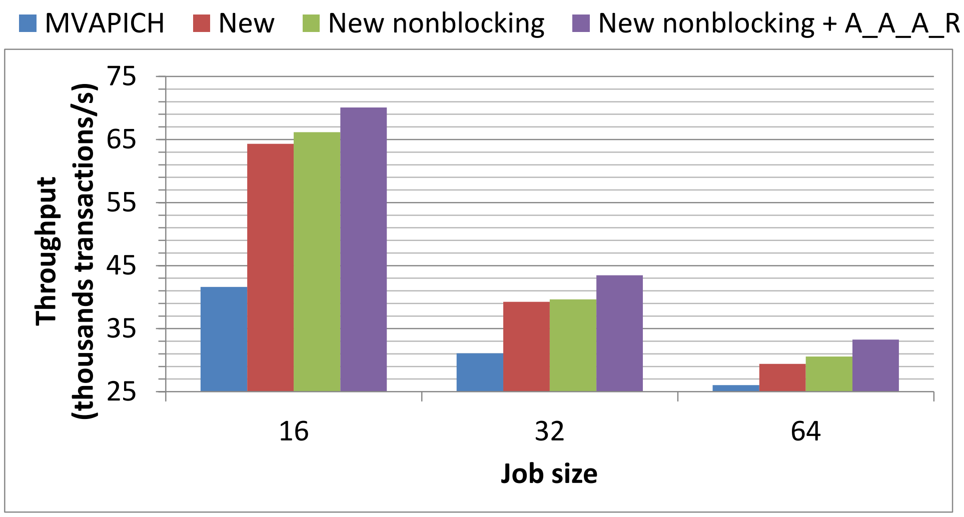
(a) Small data (1B) (b) Large data (1MB)

Figure 5.22: Out-of-order GATS epoch progression with `MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER`

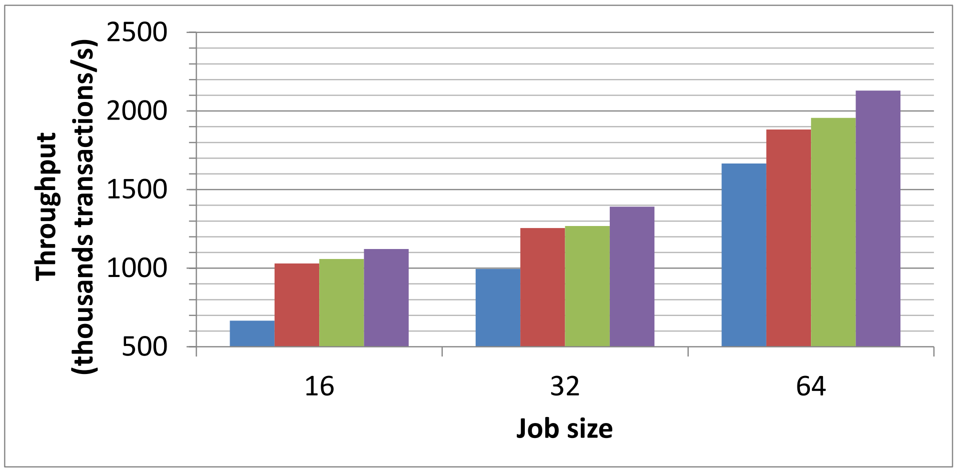
transactions for the test, the transaction size is chosen to be small so as to avoid the communications from being dominated by payload transfer latency. Each transaction updates 256B of data. Each result is averaged over 5 iterations. Four test series are considered, namely MVAPICH, New, New nonblocking and New nonblocking + A_A_A_R. The last three test series all use the new design; and the last one enables `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER` so as to allow the progress engine to aggressively progress and complete concurrent transactions meant for distinct targets. The tests are performed over 16, 32 and 64 processes (Figure 5.23).

Figure 5.23(a) shows the average transaction throughput per process; and Figure 5.23(b) show the transaction throughput for the whole job. First of all, one can notice that all the three test series based on the new design perform better than MVAPICH. However, the actual observation of interest concern how the nonblocking synchronization test series compare with the blocking synchronization ones. For a fair comparison, we restrict the analysis to the New, New nonblocking and New nonblocking + A_A_A_R test series only; as they are all based on the same design. For all the job sizes, New nonblocking performs better than New and New nonblocking + A_A_A_R performs even better. In fact, New nonblocking + A_A_A_R yields 91,972.20 transactions/s more than New for jobs of 16 processes. This difference becomes 135,058.36 and 247,230.9 transactions/s for 32 and 64 processes respectively. In terms of percentages, these three numbers respectively represent improvements of 8.93%, 10.75% and

13.14%; a trend that denotes scalability.



(a) Average transaction throughput per process



(b) Transaction throughput of the whole job

Figure 5.23: Performance of a dynamic unstructured transactional communication pattern

5.8.3 Application Test

We finally test a one-sided implementation of the Lower-Upper Gauss-Seidel solver (LU)[1] and show results over 16, 32 and 64 processes. The algorithm is entirely one-sided communications-based. Unlike the test in Section 5.8.2, no progress engine optimization is enabled for these

[1] The one-sided LU used in this chapter was developped by Xin Zhao of the University of Illinois Urbana-Champaign.

tests; the implementation is based on fence epochs. Each result is averaged over 5 executions. We show for each test:

- How much time the CPUs spent waiting for communication completion (Figure 5.24). This measurement is part ot the time that the CPUs spent inside the MPI middleware. For the blocking test series, this measurement is made of the time the job spent in the epoch-closing calls. For the nonblocking test series, the measurement is how long the job spent in `MPI_WAIT` calls.

- How much time the CPUs spent in total for communication activities (Figure 5.25). This measurement, which is a superset of the time spent waiting for communication, is the overall time spent by the CPUs inside the MPI middleware. For the blocking test series, this measurement is made of the time spent by the CPUs in all synchronization calls and all RMA communication calls. For the nonblocking test series, this measurement is made of the time spent by the CPUs in all synchronization calls, RMA communication calls and all `MPI_WAIT` calls.

- The overall CPU time spent solving LU (Figure 5.26). This measurement is made of both the time spent in communication and in computation. It is important to realize that because the LU algorithm strives to achieve communication/computation overlapping, this third measurement is less than the sum of communication and computation times. For instance, if the communication time is 60% of the overall LU solving time, the computation time could nevertheless be substantially more than 40%.

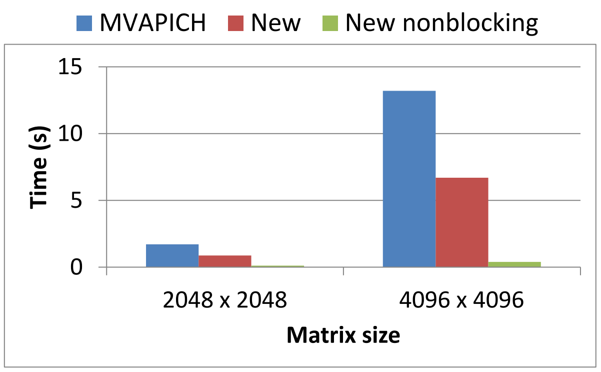
We remind that RMA calls are always nonblocking; and epoch opening fences are non-blocking in all test series as well because they resort to `MPI_MODE_NOPRECEDE`. The CPU time spent waiting for communication (Figure 5.24) and communicating in overall (Figure 5.25) are measured by PMPI-level instrumentation of the vanilla and new MVAPICH builds. PMPI [68] is an add-on interface that allows the insertion of hooks into MPI calls. Similarly to the measurements of all the previous tests, the overall LU time (Figure 5.26) is measured from inside the application.

The test setups are as follow:

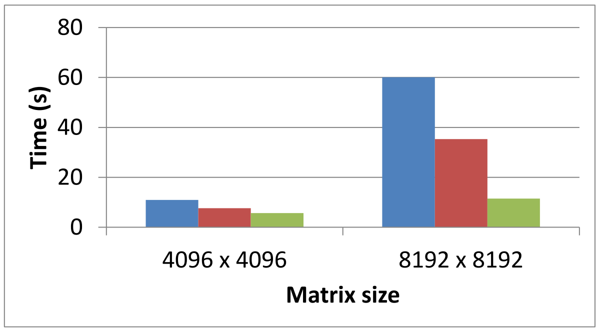
- Matrices of sizes 2048×2048 and 4096×4096 over 16 processes.
- Matrices of sizes 4096×4096 and 8192×8192 over 32 processes.
- Matrices of sizes 8192×8192 and 16384×16384 over 64 processes.

One can observe that the blocking version of the new design is better than MVAPICH for all cases; and for wait, communication and overall LU resolution times. The blocking new design being better than the vanilla MVAPICH confirms the optimized nature of the design choices made; and shows that, for the sake of realizing our nonblocking synchronization proposal, we made no compromise in what MPI-3.0 RMA already offers.

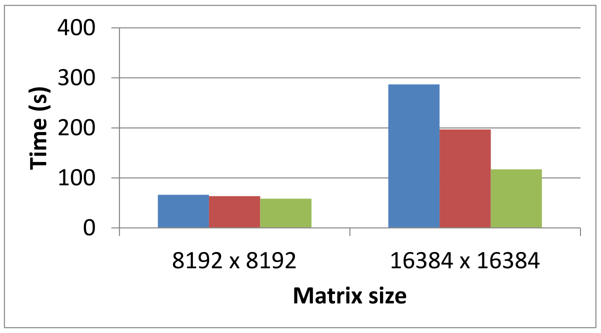
Furthermore, the comparison between the New and New nonblocking test series also shows that nonblocking synchronizations noticeably speed up wait, communication and overall LU execution times. The improvements come from the total avoidance of Early Fence and Wait at Fence inefficiencies in the New nonblocking test series. Additionally and incidentally, a higher potential for communication/computation overlapping is achieved with the nonblocking synchronizations because the RMA data transfer can be overlapped with the computation even beyond the nonblocking epoch-closing fences. LU is very computation-intensive; and the algorithm is iterative. In each iteration, all the peers open a fence epoch; and based on runtime conditions, a single peer does the data transfers of the iteration. In each iteration, all the peers compute as well; but the peer acting as origin computes before and after its RMA communications. In the nonblocking test series, no computation taking place inside the fence epochs delays the exit of the epoch-closing fence of remote peers. Additionally, the computation that takes place after the RMA transfer of the peer acting as origin can start faster because it does not wait for the epoch to complete.



(a) 16 processes

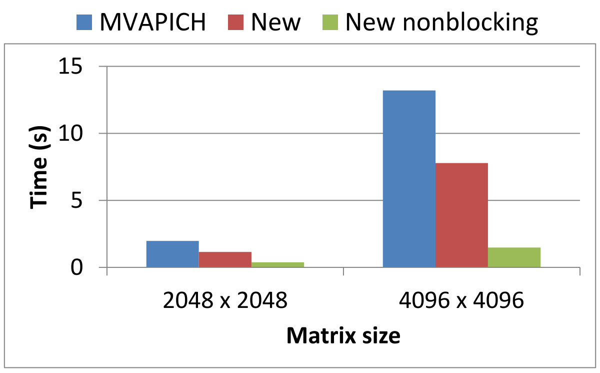


(b) 32 processes

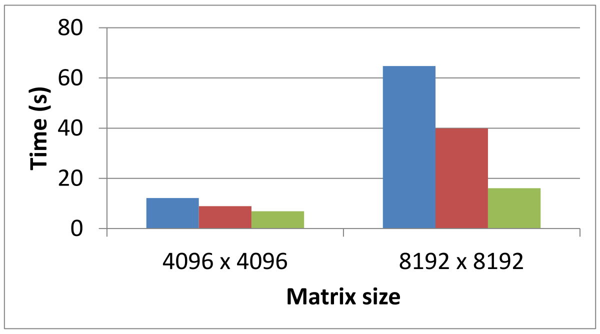


(c) 64 processes

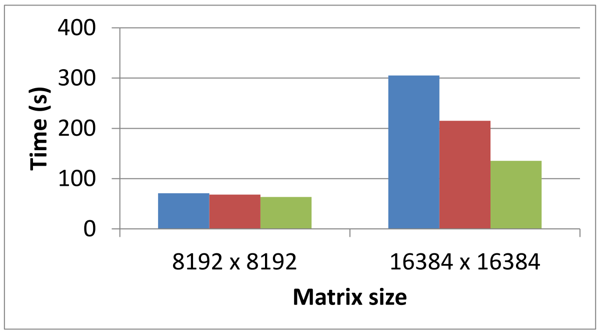
Figure 5.24: Wall-clock CPU time spent in communication wait phases



(a) 16 processes

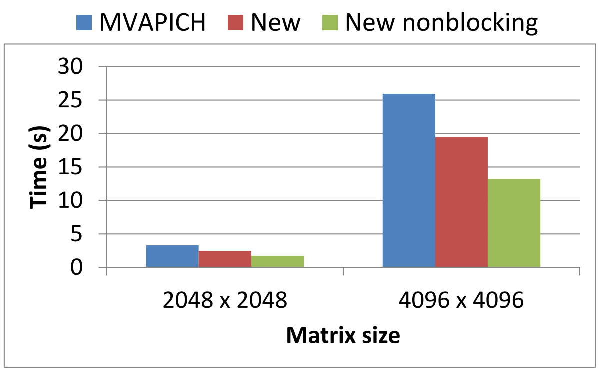


(b) 32 processes

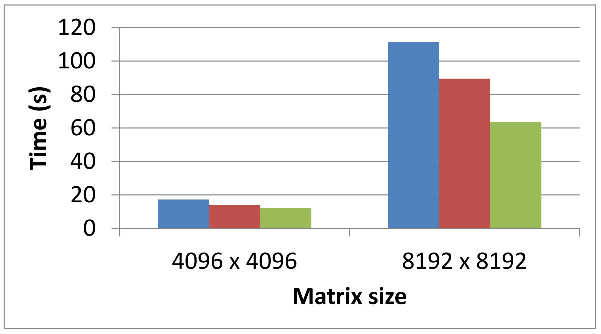


(c) 64 processes

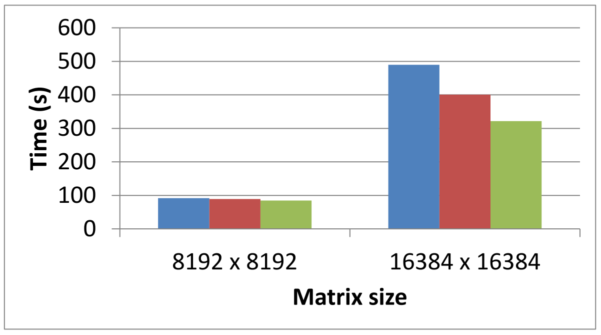
Figure 5.25: Overall wall-clock CPU time spent in communication activities



(a) 16 processes



(b) 32 processes



(c) 64 processes

Figure 5.26: Overall wall-clock CPU time spent solving LU

5.9 Summary

One-sided communications have gained importance in supercomputing for their low level of latency propagation and their consequent suitability for scalable communications. However, the one-sided communication model provided by MPI is still too constrained for the sake of correctness and memory consistency. The blocking nature of MPI one-sided communication synchronizations is not only the source of performance issues, but because it generates and propagates latency among the processes, it even defeats one of the key positive characteristics of the communication model.

The latency issues of MPI-3.0 one-sided communications are documented and categorized in six inefficiency patterns of which four are impossible to work around with the current MPI-3.0 specification. We introduce in this dissertation a new inefficiency pattern that was undocumented before. Then we propose entirely nonblocking synchronizations for MPI one-sided communications. We show that all the four inefficiency patterns as well as the newly discovered one are now solved issues with the proposed one-sided communication synchronizations. The nonblocking epochs put forth by the proposal also allow new use cases of HPC communications to be more efficiently handled in MPI one-sided communications. In particular, we show that HPC jobs with unstructured and dynamic communication patterns can now be implemented in MPI with less contention thanks to nonblocking epochs. Since nonblocking synchronizations and epochs bring additional complexities, we spell out their semantics, the hazardous situations and the behaviours to expect from the progress engine. The semantics and correctness rules are integral part of the proposal because the new design is meant to be presented to the MPI forum for consideration in a future version of the MPI standard.

Since nonblocking epochs allow the data transfer of an epoch to occur after the epoch is closed, there is an increased potential for communication/computation overlapping. At larger and larger scales, the effect of this additional serialization removal and its propagation thereof can have a significant impact on the overall execution time growth pattern of HPC applications. The proposal in this chapter has two important scalability-related impacts. First, it allows the HPC application developer to express scalable behaviours. Second, it empowers the progress engine to realize scalable behaviours. The second point is not just the realization at middleware-level of the first point that is expressed at application-level. The progress engine improvements that we propose are not limited to the passive fulfillment of nonblocking epochs; they go beyond immediate positive consequences such as increased opportunistic message progression. Thanks to the optimizations enabled by the info object flags, the progress engine can make bold decisions and reschedule the data transfers of epochs to shrink the overall communication time.

The nonblocking MPI one-sided synchronization proposal along with the resulting nonblocking epochs put forth in this dissertation are network technology-agnostic. While, our proof-of-concept implementation is based on InfiniBand, the proposals are not tied to any InfiniBand-specific feature.

In general, there are much more issues that could be associated with MPI; and they are not even necessarily communication-related. In the next chapter, we use MPI with persistent services; a type of program that exhibits regular HPC program requirements on top of being more demanding in terms of resources and failure handling. The persistent service resorts to one-sided communications to perform bulk data transfer. However, the important aspect of the next chapter is that the unusual and extreme conditions created by the persistent service allow us to investigate new important scalability aspects of MPI.

Chapter 6

MPI in Extreme-scale Persistent Services: Some Missing Scalability Features

HPC distributed services, such as storage systems, are hosted in servers that span several nodes. They interact with clients that connect and disconnect on demand; and require network transports that offer high bandwidth and low latency. These services are typically written in user space and require user-space networking APIs. However, for performance reasons, contemporary HPC systems typically employ custom network hardware and software. In order to reduce porting efforts, distributed services benefit from using a portable network API. The most likely low-level networking API for general-purpose programming is the ubiquitous 30-year-old BSD socket API. While BSD sockets are supported on HPC networks, they are not typically used because of lower bandwidth and higher latencies when compared with native networking libraries. Instead, the HPC community has seen a myriad network technologies, many of which have been short-lived. With proprietary HPC system manufacturers, in particular, there is no guarantee that an existing network API will be adopted by the next generation supercomputer. For example, the LAPI [5], DCMF [54] and PAMI [57] network APIs were all released by a single company for its Scalable POWERParallel [5] and Blue Gene [57] series of supercomputers. Unfortunately, none of those network APIs is portable across all or even most systems of their common manufacturer.

However, all recent HPC systems do include an implementation of MPI as part of their software stack. Thus, MPI shields the HPC community from the aforementioned volatility in network technologies and from low level details such as flow control and message queue management [119] (see Chapter 4). MPI has, in effect, become the BSD socket of HPC programming. Plus, since MPI is one of the primary ways of programming supercomputers, the bundled MPI implementation is typically well-tuned and routinely delivers optimal network performance [62].

Currently though, MPI is not typically used in components of the software stack that extend beyond a single application such as distributed services. Plus, while it is acknowledged that resiliency will be a major concern for exascale systems [22], nowadays' MPI applications are still tolerant to outright job abortion. Unlike application-type HPC programs whose executions are relatively short-lived, distributed services are often persistent because they bear no concept of job completion. Persistence as required in distributed services implies stricter fault handling approaches because the consequences of service abortion usually extend beyond a single job as it is usually the case in computing applications. As a consequence, distributed services inherently require from the communication substrate the forward-looking resiliency mechanisms that are anticipated for application-type HPC programs at exascale.

In this dissertation, we evaluate the use of MPI as a high-performance network portability layer for cross-application services. We proceed with an analysis of the challenges, workarounds and aspects of the design that are rendered easy, robust or difficult by the semantics of MPI. In particular, we analyze how the service design, which is resiliency-conscious and geared towards scalability with respect to the number of allowed concurrent clients, map to the features offered by MPI. Similarly, we provide various analyses which show aspects of MPI that can be improved to better serve some use cases backed by the service design. We go beyond the sole resiliency concerns to reveal other MPI scalability weaknesses that application-type HPC programs do not bring up [120, 121]. The other uncommon use cases provided by the distributed service create conditions of internal MPI object depletion, resource exhaustion, communication cancellation and non-communicating routine scalability needs. Our observations make the case for feature additions or improvements in MPI. We argue that these feature proposals are timely and are even relevant to mainstream MPI-based HPC applications on future machines.

6.1 Related Work

MPI has been used in a few previous work outside the usual HPC computational programs. A study of the possibility of MPI adoption for the storage architecture of PVFS2 and parallel persistent services in general was presented in [60]. The study stated that MPI could be used for a broader range of parallel utilities such as system monitoring deamons. In particular, MPI has been used for file staging and parallel shell design [18]. The I/O delegate proposal [77] allows an MPI job to transit its I/O requests through another MPI job linked to the target filesystem. It resorts to dynamic process management, but it is not a pure client-server design. The compelling aspects of MPI have attracted other data-oriented distributed services and runtimes as well. MapReduce, for instance, has been studied and layered on top of MPI [40, 62, 82]. Moreover, the PGAS languages have considered MPI for its wide adoption, performance, and richness of programming models [10]. Except for the PVFS2 study [60], none of the cited works were examples of unrelated MPI jobs linked by a client-server relation; and to the best of our knowledge, no implementation of PVFS2 over MPI exists yet.

6.2 Service Overview: The Distributed Storage

As a concrete example of distributed services, this section presents the highly-available distributed storage system used as discussion support in this chapter. The storage system, depicted in Figure 6.1, is made of I/O servers which export a unified view of the underlying storage to a set of clients. The clients are mostly compute nodes running application software. Clients typically connect to the storage service at the start of a job, periodically issue I/O requests, and disconnect when the job ends. In our system, the storage service relies on replication to ensure that data remains available even when an I/O server fails. In addition, data is striped across multiple servers, increasing performance by providing parallel access to the underlying storage devices. The client is oblivious to striping and replication; the I/O library on the client side only exposes traditional read/write semantics.

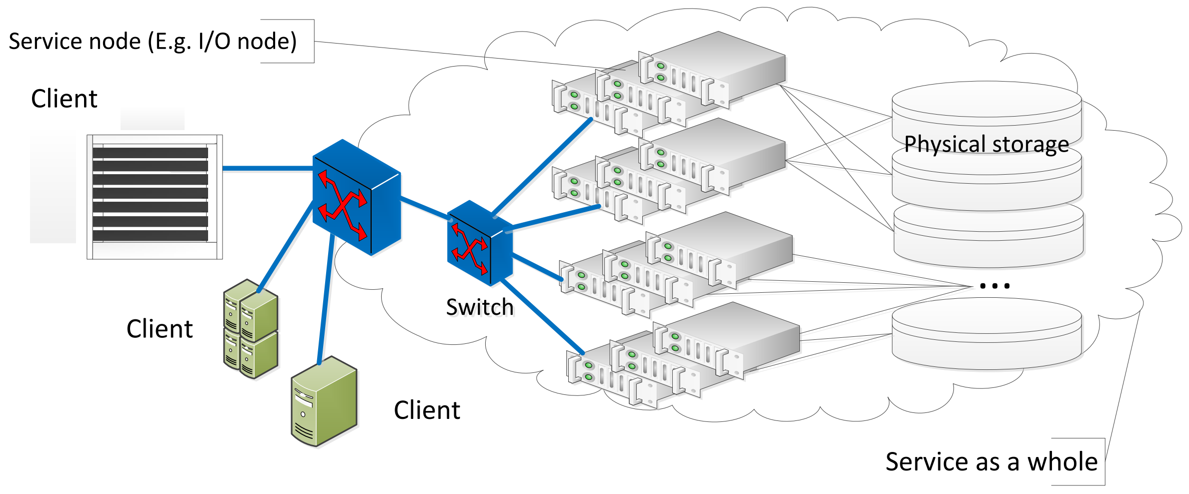
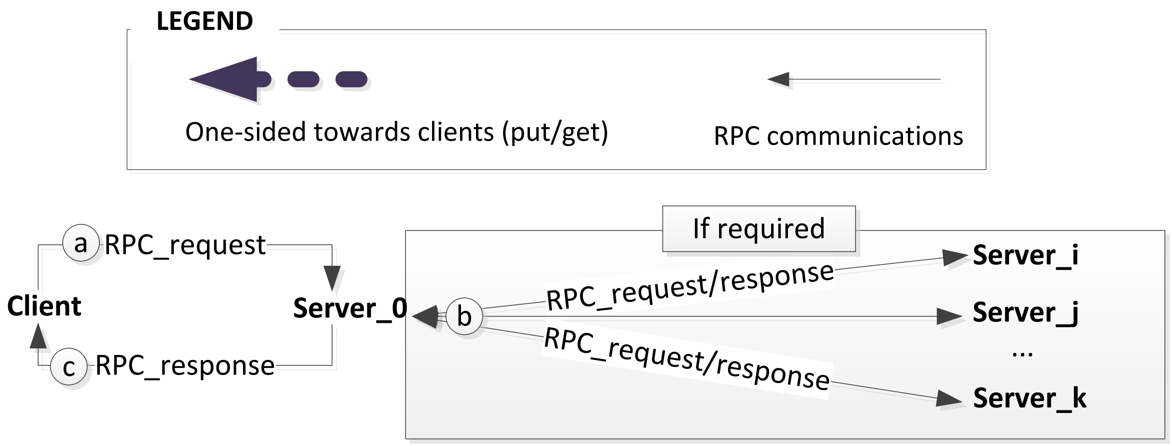


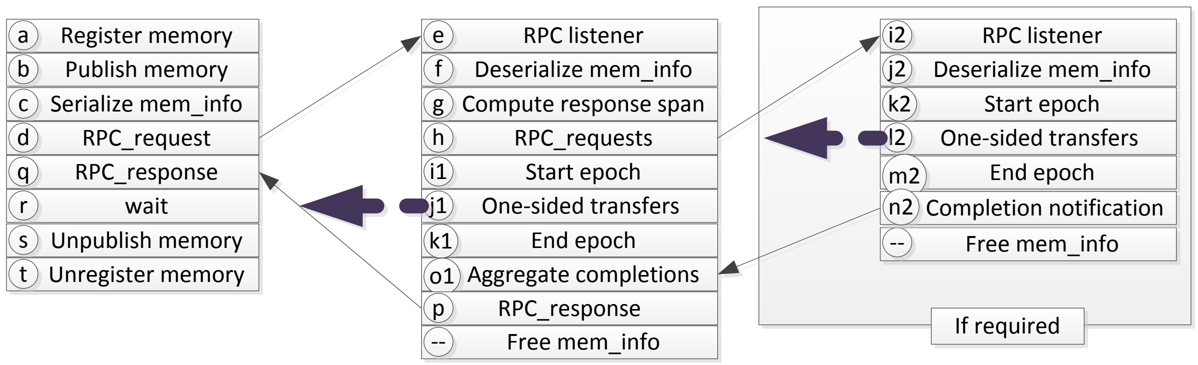
Figure 6.1: Distributed storage service

All the I/O requests start with a Remote Procedure Call (RPC) initiated from a client to a single server which executes the request and then sends a response back to the client. Because of striping, a single, sufficiently large I/O access can span several I/O servers. However, to limit client-side complexity and simplify failure handling, a client always contacts a single server for any I/O request. That server manages striping and replication on behalf of the client and provides a single response for client-side I/O requests. The contacted server relays the request of the client to any secondary server involved, for purposes of striping or replicas, and aggregates the response of each server before responding back to the client. The single server contacted by a client is not fixed; it is decided by a placement policy running in the upper layers of the storage system. The placement policy can lead the same client to contact two distinct I/O servers for two distinct I/O requests even if they both target the same data. For any given I/O request, we designate the contacted I/O server as primary server or server_0; all the other involved servers are referred to as secondary servers.

We distinguish between small and large I/O payloads. A small payload usually resides on a single physical storage managed by a single server; in which case, its I/O lifetime includes only steps a and c of Figure 6.2(a). When a non-contiguous small payload has large enough gaps in its spanning address to reside on several physical storages, it can involve several I/O servers. Server_0 in that case must seamlessly scatter (gather) the relevant chunks of data to (from) the other concerned servers; as shown in step *b* of Figure 6.2(a).



(a) Small payload



(b) Large payload

Figure 6.2: I/O transfer protocol

For large messages however, the intermediate copies associated with the use of server_0 as payload router can be noticeable. The protocol in Figure 6.2(b), meant for bulk data transfers, is therefore used to allow multiple servers to one-sidedly copy chunks of the I/O payload into or from the client's memory. The lifetime of a large-payload I/O starts with the client registering an I/O buffer in step *a*. A subset or the entire registered buffer is then made available for remote access in step *b*; the buffer is said to be published. The distinction between buffer registration and publishing is a performance-conscious choice for network transports where a single registration cost can be amortized over multiple I/O. As a consequence, if the buffer or any of its spanning address supersets was already registered, the lifetime of a large-payload

I/O could start at step *b* and ends at step *s*. Step *t* undoes registration. Ideally, registration does not grant remote access. Remote access granting starts with publishing in step *b*. Then, in step *c* the remote access information pointed to by the publishing handle is serialized into the RPC packet meant to initiate the I/O. The access information is not tailored for a specific remote server. As a consequence, server_0 can resend it to any other servers involved in the associated I/O, so that they could seamlessly one-sidedly access the client.

The I/O request sent by the client in step *d* is caught by the RPC listener of server_0 in step *e*. The server reproduces the memory access information in step *f*; and in step *g*, determines all the other (secondary) servers which must be seamlessly involved in fulfilling the I/O. Server_0 involves the secondary servers by dispatching the new intermediate RPC requests (step *h*). Then in steps *i1*, *j1* and *k1*, server_0 transfers its chunk of I/O payload to or from the client. In parallel, the secondary servers fulfill steps *i2* to *m2* to transfer their respective partitions of the I/O payload. Server_0 blocks in step *o1* until it receives a completion acknowledgement from each of the involved secondary servers. Server_0 sends a final RPC response to the client in step *p*.

For certain network transports, the completion of step *q* at the client side could guarantee that the whole I/O payload has been transferred. However, since the one-sided payload transfer of the secondary servers and the RPC response which contains their completion acknowledgement have different destinations, they might complete out-of-order; leading to the final reply from server_0 to the client potentially completing before some payload transfers. For network transports where such an out-of-order completion might occur, a custom safe completion subprotocol can be implemented by sending the right control data in the RPC response in steps *p*-*q* and by checking it in step *r*. Step *r* is blocking. After the memory region is unpublished in step *s*, it is expected to become inaccessible to remote servers.

The I/O protocol is network-agnostic. It is therefore generic enough for certain steps to be either redundant or irrelevant for certain network transports. Redundant or non-applicable steps exit immediately because they are implemented as noop routines. For instance, step *r* of Figure 6.2(b) is a noop in an MPI implementation of the protocol because the one-sided transfers are fulfilled in passive target epochs. Consequently, by the time they complete at the server, they are guaranteed, as per the MPI specification, to have completed in the client's memory as well. The acknowledgement of each server is thus guaranteed to be issued only after its payload chunk has entirely reached the client.

The protocol presented in Figure 6.2(b) reduces the client-side protocol complexity since the client is not logically involved in the transfers. As the client is oblivious to all the secondary servers, the protocol simplifies the handling of client and server failure. It also defers flow-control to the I/O server and eases the burden of high ratio of compute nodes over I/O nodes in current and future HPC systems. The work on Mercury [97], which is centered around RPC over the same protocol, provides in-depth comparisons with other existing RPC frameworks and protocols.

6.3 Transport Requirements

The set of network features required by the service are as follows.

Connection-disconnection: The network transport must be able to efficiently handle the random addition and removal of clients.

Autonomous data transfer: All network operations need to support autonomous data transfer. Host threads cannot be dedicated to message progression because they are a scarce resource in the server nodes.

Two-sided communications: RPC invocations are built on top of a two-sided communication semantics. A tag mechanism is required to differentiate among messages between the same two peers. The tag space must be big enough to render message collisions unlikely by preventing any service-level message from reusing a tag currently being used.

One-sided communications: For the client to be effectively oblivious of stripping, replication and the secondary servers in particular (Figure 6.2(b)), it has to be a passive target in a true one-sided communication scheme.

Scalability: The scalability of a distributed service is determined, among other things, by the scalability of its underlying network. In the particular case of this storage system, the network is expected to allow a large number of simultaneous clients; each potentially having multiple concurrent I/O requests.

Failure mitigation: The failure of an isolated server node should not halt the service. Replication actually makes every server hot-replaceable; and the network transport is expected to ease the replacement by keeping the healthy servers both alive and connected. Additionally, the service should not be brought down by a failed client; as many other clients are potentially connected to the servers at any given time. A failed client should also not impact other unrelated clients connected to the same servers. For the service to proactively deal with these resiliency concerns, some support is required from the network to detach failed nodes without global impact on the healthy server nodes. Additionally, when a client or server process stops responding, their targeting peers should be able to cancel any pending network operation. For clients, the cancelled operation can be retried to a different healthy or more available server. For servers, resources can be freed and redirected towards other healthy clients.

6.4 MPI as a Network API

In this section, we provide a bit of implementation detail in order to clearly show why certain aspects of MPI are subject to the call for improvements that we introduce later in the chapter. As MPI is an interface, it is important to show how any new API addition stemming from a recommendation can be used; but first, it is useful to show how any current routine that we seek to improve with a new proposal is used in the current API of MPI.

When running over MPI, the storage server is an MPI job created in `MPI_THREAD_MULTIPLE` mode. The MPI-based server can only interact with other MPI jobs as clients. As a consequence, even a client made of a single standalone process must be a single-process MPI job. Each server node hosts exactly one process of the overall distributed storage job. In order to handle large numbers of simultaneous requests in a scalable manner, each I/O process has:

- A dedicated connection thread which listens on a connection channel and executes the procedure required for first contacts from soon-to-be clients.
- A dedicated listener thread which receives all the requests from already existing clients and transforms them into a work item which is enqueued for a threadpool to handle.

This thread fulfils the RPC listener.

- A threadpool made of non-dedicated worker threads which handle the potentially heavy tasks from the work queue. Unless otherwise specified in a configuration file, the size of the threadpool on any node is exactly $n - 2$ if the node can run simultaneously n hardware threads. Oversubscribing is thus avoided.

The two dedicated threads do very little except for receiving connection and RPC requests; the goal being to be as reactive as possible to concurrent requests.

We present observations of certain limitations of MPI or MPI implementations in this section. The executions are done with OpenMPI-1.6.5, MPICH-3.0.4, and MVAPICH2-1.9. MPICH is the only implementation which can run the fully-fledged server, as none of the other two MPI distributions provides simultaneously a working implementation of MPI-3.0 one-sided, a support for `MPI_THREAD_MULTIPLE` and the client-server connection features of MPI. However, we did observe the behaviour of OpenMPI and MVAPICH for isolated requirements of the design. Our first test system (C1) is a 4-node cluster with each node having two quad-core 2GHz AMD Opteron 2350 processors and 8GB of memory. The second system (C2) is a 310-node cluster with each node having two Nehalem 2.6 GHz Pentium Xeon CPUs with hyperthreading disabled; and 36 GB of memory. Both systems have Mellanox ConnectX QDR InfiniBand and Gb Ethernet. The following subsections describe how specific aspects of the client-server orchestration are fulfilled in MPI.

We emphasize that the tests performed in this chapter are not performance evaluation. These tests are the basis for various observations made in the chapter. The tests are consequently not grouped in a dedicated single section as it is the case for the previous chapters; instead, they are performed in context, where required, to allow the specific observation they are meant for.

6.4.1 Connection/Disconnection

At initialization time, the storage server opens a port with `MPI_OPEN_PORT`, and then the connection thread of each individual server process waits in `MPI_COMM_ACCEPT`. No server spawns any client and no client spawns any server. In fact, the service is expected to be already running by the time any client attempts connection; and of course, it wouldn't make sense for the service to spawn its clients. In a multi-process MPI program, the job as a whole could create a single I/O connection; but any subset of its processes could create distinct I/O connections as well. When a set of processes connect as a single client group, they must also disconnect as a whole. The following MPI functions are used to fulfill the connection/disconnection procedures:

- `MPI_COMM_DUP`: The routine duplicates a communicator. The child communicator has the exact same characteristics as the original but its communications are isolated from the ones of its parent.

- `MPI_INTERCOMM_MERGE`: This routine converts an intercommunicator into an intracommunicator.

- `MPI_COMM_CONNECT`: This routine emulates the connection mechanism of BSD sockets. It allows an unrelated job acting as a client to connect to another MPI job acting as a server.

- `MPI_COMM_ACCEPT`: This routine is the server-side counterpart of `MPI_COMM_CONNECT`. It allows an MPI job to accept incoming connections from other MPI jobs.

- `MPI_COMM_DISCONNECT`: This routine undoes a connection created by either `MPI_COMM_CONNECT` or `MPI_COMM_ACCEPT`.

- `MPI_WIN_CREATE_DYNAMIC`: This routine creates an RMA dynamic window. Unlike regular RMA windows created with `MPI_WIN_CREATE` for instance, the RMA buffer of a dynamic window does not have to be defined once for all at creation time. Any number of RMA buffers can be attached to and detached from a dynamic window during its lifetime.

- `MPI_WIN_LOCK_ALL`: This routine, which was briefly described before, allows a process to lock all the peers of an RMA window in a passive target epoch.

- `MPI_WIN_UNLOCK_ALL`: This routine undoes `MPI_WIN_LOCK_ALL`.

- `MPI_WIN_FREE`: This routine frees the RMA window object.

The client connection routine is shown in Listing 6.1. At the server side, the connection thread blocks on `MPI_COMM_ACCEPT` in a loop which breaks out only when the server enters termination mode. Every time `MPI_COMM_ACCEPT` returns, a work item with the handler shown in Listing 6.2 is queued to be executed by the threadpool.

```
client_connect(MPI_Comm comm, client_connection_t** conn_obj)
{
  read_port(__OUT__ port); //read server connection port
  *conn_obj = allocate(...);
  MPI_Comm_dup(comm, &(*conn_obj)->app_comm_dup);
  MPI_Comm_connect(port, ... , (*conn_obj)->app_comm_dup, &(*conn_obj)->
    connect_comm);
  MPI_Intercomm_merge((*conn_obj)->connect_comm, FALSE, &(*conn_obj)->io_comm);
  MPI_Win_create_dynamic(..., (*conn_obj)->io_comm, &(*conn_obj)->win);
  /*... other irrelevant instructions ...*/
}
```

Listing 6.1: Client-side connection code snippet

```
server_connect_work_handler(work_t* work)
{
  server_connection_t* conn_obj = allocate(...);
  conn_obj->connect_comm = get_comm_from_work_item(work);
  MPI_Intercomm_merge(conn_obj->connect_comm, TRUE, &conn_obj->io_comm)
  MPI_Win_create_dynamic(..., conn_obj->io_comm, &conn_obj->win)
  MPI_Win_lock_all(0, conn_obj->win)
  enqueue_new_connection_obj(conn_obj)
  notify_request_listener_for_new_client()
  /*... other irrelevant instructions ...*/
}
```

Listing 6.2: Server-side connection code snippet

`MPI_COMM_CONNECT` and `MPI_COMM_ACCEPT` return an intercommunicator. However, an intracommunicator is required to create the one-sided communication window; hence the need to invoke `MPI_INTERCOMM_MERGE` to create the I/O communicator `io_comm`. After the client is connected, RPC and I/O occur over `io_comm` communicator of the connection object.

```
 1 client_disconnect(client_connection_t* conn_obj)
 2 {
 3   clean_zombie_receives(conn_obj);
 4   MPI_Barrier(conn_obj->app_comm_dup);
 5   notify_server_for_disconnection(); //This is an RPC
 6   MPI_Win_free(&conn_obj->win);
 7   MPI_Comm_free(&conn_obj->io_comm);
 8   MPI_Comm_disconnect(&conn_obj->connect_comm);
 9   MPI_Comm_free(&conn_obj->app_comm_dup);
10   free(conn_obj);
11 }
```

Listing 6.3: Client-side disconnection code snippet

```
 1 server_disconnect_work_handler(work_t* work)
 2 {
 3   conn_obj->connect_comm = get_comm_from_work_item(work);
 4   clean_zombie_receives(conn_obj);
 5   MPI_Win_unlock_all(conn_obj->win)
 6   MPI_Win_free(&conn_obj->win);
 7   MPI_Comm_free(&conn_obj->io_comm);
 8   MPI_Comm_disconnect(&conn_obj->connect_comm);
 9   free(conn_obj);
10   free(work);
11 }
```

Listing 6.4: Server-side disconnection code snippet

The client-side disconnection executes the procedure shown in Listing 6.3. `clean_zombie_receives` (line 3) is described in Section 6.4.4. A barrier is required (line 4) to ensure that all the processes in the client group have entered their disconnection routine before the server is contacted. Since the server does not actively wait for clients to disconnect, it has to be notified over the RPC listening channel. A process in the client groups sends the notification at line 5 of Listing 6.3. The rest of `client_disconnect` simply undoes what was created in `client_connect`. After `client_disconnect` exits, the client job becomes totally detached from the server.

At server side, after it receives the disconnection request, the RPC listener removes the concerned connection object from the list of endpoints to listen on; and enqueues a threadpool work item with the handler shown in Listing 6.4. At the exception of `clean_zombie_receives`, `server_disconnect_work_handler` mostly undoes what each server process did to create the connection object in the first place.

6.4.2 The RPC Listener

In each server node, the RPC listener is a dedicated thread which listens to RPC requests coming from either the clients or from the other servers. Since each connection object has its own I/O communicator (`io_comm` at line 7 of Listing 6.1), the RPC listener must post one `MPI_IRECV` per connection object. All the `MPI_IRECV` of the listener are posted with `MPI_ANY_SOURCE` and `MPI_ANY_TAG` so as to allow the server to accept unexpected requests from unknown sources. With all these pending receives, the listener waits in `MPI_WAITSOME` and gets activated only when at least one request has reached its hosting server node. `MPI_WAITSOME` is a blocking communication completion routine described in Section 2.1.5. The listener decodes the arrived requests returned by `MPI_WAITSOME`. Out of each received request, the listener makes a work item that is appended to the queue of the threadpool. Before going back to waiting in `MPI_WAITSOME`, the listener posts a new `MPI_IRECV` for each of the connection objects on which a request has just arrived; so it could continue listening to the clients associated with each of those objects. However, this step is skipped for requests asking for disconnection.

At initialization time, the RPC listener listens on only one connection object which is the channel where the server nodes talk to one another inside the service. Then, after any new connection is created, the connection thread sends a `NOTIFY_SELF` request to its own node (line

9 of Listing 6.2); allowing the listener to get out of `MPI_WAITSOME` and add the new connection object to its list of client groups to listen to.

6.4.3 I/O Life Cycles

RPC and MPI Two-sided Communications

RPCs are central to the I/O life cycle; so it helps to describe them a bit more in depth before going any further. RPCs are the main service-level primitive and are implemented on top of MPI two-sided communications. Each RPC is a round-trip communication made of a request and a response. For performance reasons, we ensure that the size of an RPC never exceeds the threshold of MPI Eager protocol message sizes. The RPC header bears the address of the sending endpoint; in this case, the rank of the process. A tag mechanism is used to distinguish between multiple simultaneously pending requests from the same client. The tag is a service-level concept but it trivially maps to MPI tags as used in two-sided communications. The service has to guarantee that requests never collide; as a consequence, the tag space for each connection object is expected to be big enough to be considered infinite during the lifetime of a typical client. The MPI standard defines the valid per-communicator tag space to be 0..`MPI_TAG_UB`, where `MPI_TAG_UB` is required to be at least 32,767. Our test indicates values of $2^{31} - 1$, $2^{30} - 1$, and $2^{31} - 1$ for OpenMPI, MPICH and MVAPICH respectively, which make sufficiently big tag spaces. Communications between servers must use the RPC route as well in order for all control message communications to benefit from the resiliency policies implemented in the service. The resiliency policy is uniformly implemented at RPC level; on top of any transport, such as MPI, that the service is running on.

Bulk Data Transfer and MPI One-sided Communications

The MPI implementation of the memory registration/deregistration of the bulk data transfer protocol (steps *a* and *t* in Figure 6.2(b)) are noop. Only memory publishing/unpublishing (steps *b* and *s*) is implemented, via `MPI_WIN_ATTACH` and `MPI_WIN_DETACH`. At the server-side, the one-sided payload transfers resort to passive target epochs and request-based one-sided communications (`MPI_RPUT` and `MPI_RGET`). The *Start epoch* step of the protocol does not map to any MPI equivalent as an MPI epoch was already open over the clients at connection time. As for the *End epoch* step, it maps to testing at MPI-level the completion of the requests returned by `MPI_RPUT` and `MPI_RGET`. In the protocol, *End epoch* (steps *k1* and *m2* in Figure 6.2(b)) is supposed to be blocking. In the actual design, the worker threads cannot afford to block; even in portions of the protocol, such as steps *k1*, *o1* and *m2* where the protocol dictates a wait. In general, we achieve any blocking step of the protocol by reposting in the threadpool its associated work item with the same handler. The handler determines the step, thus, there is no advancement in the protocol as long as the handler of the next step is not attached to the work item. This approach allows a few worker threads to handle any number of I/O. As every request-based blocking completion routine of MPI (e.g. `MPI_WAIT`) has a nonblocking equivalent (e.g. `MPI_TEST`), designing a nonblocking handler for blocking steps was straightforward.

The design of the I/O protocol predates the availability of the MPI-3.0 specification. As a result, we had previous implementations based on MPI-2.2 RMA and on two-sided emulation of the one-sided communications [97]. MPI-3.0 substantially alleviates the challenge encountered with MPI-2.2.

In MPI-2.2, the **constraint of a single epoch per one-sided window**, the **collective nature of window creation** and the **impossibility of completing a one-sided communication without closing an active epoch** rendered the design very challenging. These constraints limited how many concurrent I/O the same client process can have pending in MPI-2.2, as each bulk data transfer required its own separate RMA window. Unfortunately, creating these windows on demand would lead to having collective calls in the middle of each large-payload I/O; defeating the purpose of hiding the layout of the storage to clients, on top of hindering performance. As a result, a certain number of one-sided windows was created at connection time; and then used by the server as flow-control credit for the maximum number of concurrent I/O to handle per client. With the new MPI-3.0 specification, request-based RMA routines allow any number of distinct I/O to complete in a nonblocking way in the same RMA epoch; voiding the need for multiple epochs or windows for concurrent I/O. Furthermore, the new `MPI_WIN_LOCK_ALL` allows the same RMA epoch to target all the processes in a client group. In summary, a single RMA window can now permit any number of concurrent I/O for a whole client group. Even though window creation is still collective in MPI-3.0, it is no longer a burden in the design.

Furthermore, **memory management** in the bulk data transfer protocol was an issue with MPI-2.2. We had to preallocate a large pool of memory on each client and resort to complex custom allocators to fulfil buffer registration and publishing at service-level (steps *a*, *b*, *s*, and *t* of Figure 6.2(b)). By resorting to MPI-3.0 one-sided dynamic windows, we avoid the fixed preallocated buffer. Memory publishing can now be performed on the fly with `MPI_WIN_ATTACH`.

At the time of doing this work, there was no known working and reasonnably bug-free open-source MPI-3.0 one-sided implementation over RDMA-enabled or autonomously progressing networks such as InfiniBand. However, the performance results of this RPC-based I/O protocol over two-sided emulation of MPI one-sided routines are available for Cray interconnect and InfiniBand in [97].

6.4.4 Cancellation

No RPC, and consequently no I/O, is allowed to block forever. As a result, all RPCs bear a timeout after which they must expire. As a reminder, a client-side expired RPC is retried, possibly to another server decided by the placement policy. A server does not retry expired RPCs; it relies on clients to re-initiate the whole operation which leads to the server issuing the RPC in the first place.

MPI_CANCEL and Cancellation Outcomes

The RPC cancellation implementation is over MPI's own cancellation mechanism, namely, `MPI_CANCEL`. `MPI_CANCEL` has an atomic behaviour with respect to a two-sided communication. The communication is either entirely cancelled or entirely completed (page 72 of the MPI-3.0 specification [68]). From the sender side, the transmission finishes if it has started before the cancellation becomes effective. At the receiver side, the destination buffer either receives the whole data or is untouched. This behaviour is welcome in the service design because it considerably eases the management and analyses of post-cancellation consistency in the service. The only important information a server holds about clients is the connection objects. This choice favours resiliency by making any server easily replaceable. It is not harmful to have temporary I/O objects in inconsistent state inside server processes when an I/O is cancelled; servers discard or recycle those objects without danger upon cancellation at their own level. In summary, the objects whose consistency state matter in situations of cancellation are the initial and final payload buffers. In particular, we need to care about the client-side buffer and the storage buffer in each server. The intermediate buffer that server_0 uses for small I/Os (Figure 6.2(a)) is a disposable object; its state becomes irrelevant as soon as the I/O it is being used for is completed, successfully or via cancellation. Cancellation for any RPC which, like disconnection requests, do not involve any payload is always without consequence.

When an RPC is cancelled by its initiator, both the request which maps to `MPI_ISEND` and the response which maps to `MPI_IRECV` are cancelled. From the other end, cancellation, if initiated by the replier, maps to cancelling `MPI_ISEND` only. The RPC-level cancellations can lead to four immediate MPI-level outcomes:

1. **Effective at request transmission**: The cancellation occurs before the request is even sent by MPI. In this case, both the send and the receive are effectively cancelled at MPI level. If the request was meant to initiate an I/O, the I/O is never seen by any server. Any subsequent retry occurs as if the previously cancelled I/O never existed. This cancellation outcome can occur if the request takes longer than required to be sent. Since RPCs use Eager data transfers at MPI level, this scenario is rare. However, with certain MPI implementations such as MVAPICH, a flow control mechanism puts a cap on the number of unmatched messages that each process can send to a given peer. As a result, an Eager send from a given client process to a specific server can be delayed if the server already has an important backlog of receives to retrieve from the same client.

2. **Effective at reply reception**: The cancellation occurs after the request is sent but before the reply reaches the RPC initiator. In this case, only the receive is effectively cancelled. Since the request has been already sent, the reply arrives anyways. With no

more receive to consume it, it sits at the receiver side inside the MPI middleware message queue. We name those messages *zombie receives*.

3. **Effective at reply transmission**: This scenario always occurs inside a server. If the server RPC was created as an intermediate one to fulfil an initial client request, then the client RPC ends up timing out as well and gets cancelled.

4. **Ineffective**: The cancellation is issued too late to be effective on any send or receive. The immediately initiating RPC completes successfully. If that RPC is an intermediate one, then the initial client RPC can end up completing as well.

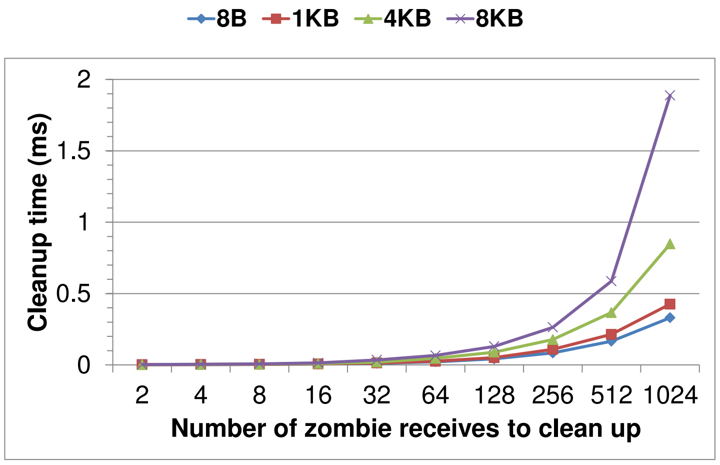
Cancellation can then lead to two kinds of undesirable consequences, namely **zombie receives** and **partially modified buffers**. Persistent partially modified buffers are very infrequent. As long as a successful client-side retry ends up happening for an RPC which got cancelled, the final buffer meant to be written into, or at least an equivalent of that buffer, ends up getting exactly the expected data. For a large-payload read, the client-side buffer could be left partially modified by cancelled RPC between the servers. Transfers between two peers is still subject to the all-or-nothing pattern but all servers might not transfer their partitions of the payload if some of them perform any kind of effective cancellation. In this case however, the client buffer ends up being overwritten by the full payload upon the first successful retry of the initial requester. Then, the buffer remains unchanged even if it is overwritten by portions of the same data by any pending cancelled I/O. For a write, a retry can involve replicas, in which case the same server-side buffer is not modified but its equivalent in a replica server is; leading to the last stored data being the right one.

Dealing with Zombie Receives

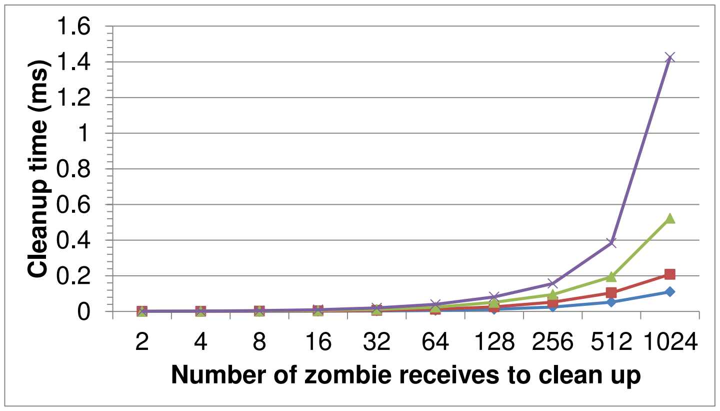
A zombie receive is essentially an MPI-level unexpected message queue item [119] which is abandoned forever. It could also be sitting inside the network device, waiting to be extracted by the MPI middleware. The tag management guarantees for a given client that any response to a cancelled RPC will no longer be matched inside MPI for approximately `MPI_TAG_UB` subsequent RPC requests, failed ones included. On top of the resource consumption issue, this situation can create the following confusion in the servers. Message matching in MPI identifies a communicator by its contextId (Chapter 4). When a communicator is destroyed in a server after the disconnection of a client group C_i, MPI can recycle and reuse its contextId for a new communicator created for a subsequent client group C_j. Then, a server can mistakenly consume zombie C_i requests as if they were sent by the new C_j client group. In order to avoid that situation, all the zombie receives must be matched and removed from inside the MPI middleware buffers before the destruction of their associated communicator in the disconnection routines.

For a connection object, it is challenging to safely spot the zombie receives from legitimate ones as long as the client is still issuing RPCs. However, right after the disconnection request is received by the servers and the reply reaches the client, each end has the guarantee that no more MPI-level two-sided communication will arrive over the connection object about to be disconnected.

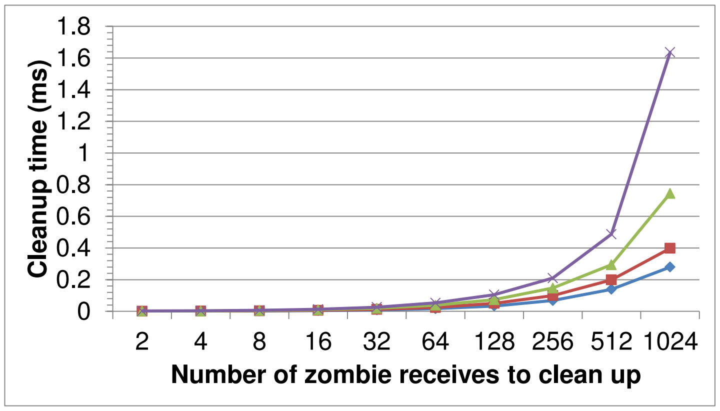
In a loop, each zombie receive is discovered via `MPI_IPROBE` with `MPI_ANY_SOURCE` and `MPI_ANY_TAG` and matched with a dummy `MPI_RECV`. `MPI_IPROBE` is a nonblocking routine that peeks into the MPI middleware to check for the arrival of a two-sided message without retrieving it. The cleanup stops when no more items are found by `MPI_IPROBE`. The cleanup has a de facto local semantics. Since all the cleaned up receives were messages already sitting locally inside the MPI middleware, no network communication is involved. Figure 6.3 shows the latency of zombie receive cleanup for RPCs made of a small control message (8B) to some pretty large buffers containing embedded I/O payload (up to 8KB). Each test is performed over 100 iterations on the system C2. As shown in Figure 6.3(a), Figure 6.3(b) and Figure 6.3(c), less than 2 milliseconds is required to remove up to 1024 zombie receives even when there is a payload of close-to 8KB. In comparison, the average rotational latency of the fastest mechanical hard drive in existence to date (15,000 RPM) [113] is 2 milliseconds for a single access. Assuming that 1) no peer is abruptly killed by any unforeseen fault and 2) no MPI one-sided communication is involved in servicing the RPCs of interest, these results show that there is an insignificant penalty to managing the only negative consequence of resorting to `MPI_CANCEL` for RPC resiliency.



(a) MPICH



(b) MVAPICH



(c) OpenMPI

Figure 6.3: Zombie receive cleanup latency

Concluding Remarks on MPI Adequacy for Service-level Cancellation

Cancellation as provided by MPI for two-sided communications is adequate to fulfil the needs of the service. It is worth mentioning that `MPI_CANCEL` helps fulfilling a key scalability trait of the service because it always exits immediately; potentially before the actual cancellation occurs. As a result, clients can, in a timely fashion, issue retries to other potentially more available servers and take advantage of replication whenever possible.

Unfortunately, the situation is different for one-sided communications. Once an I/O request times out and its associated RPC gets cancelled, the protocol (Figure 6.2(b)) requires the access to the client buffer to be revoked for the cancelled I/O. Service-level buffer access removal maps to `MPI_WIN_DETACH`, which cannot be safely invoked until any one-sided data transfer targeting the attached buffer is completed. Furthermore, one-sided communications cannot be cancelled at MPI level; leading to the impossibility of deallocating resources in a timely fashion. For any given peer, such a situation can become unmanageable when any of the involved remote peers becomes indefinitely unavailable due to a fault for instance. In conclusion, MPI provides none of the one-sided communication-related mechanisms required for implementing the resiliency policy of the service.

6.4.5 Client-server and Failure Behaviour

By default, MPI aborts a job when an error occurs inside the middleware. This behaviour corresponds to the `MPI_ERRORS_ARE_FATAL` mode. MPI also defines `MPI_ERRORS_RETURN` as another error mode where failing MPI routines return an error code to the application instead of aborting. The error mode is specified per communicator or per RMA window.

The storage server always runs with `MPI_ERRORS_RETURN` set for communicators and windows. We consider two kinds of failures: abort and crash. In the abort failure, the faulty process is still alive but behaves inappropriately. It must leave and rejoin the storage system after its problem is fixed. Thus, after the possible cleanup, the process terminates with `MPI_ABORT(MPI_COMM_SELF)`. `MPI_ABORT` aborts all the processes of the communicator it is given as argument. The crash failure, simulated with a provoked segmentation fault, is a brutal death caused by a node shutdown, for instance. The tests in this subsection are run on the cluster C2; with a single process per node. The experiments described in this section test isolated features or their equivalents. For instance, because MPI-3.0 one-sided is not supported by all three MPI implementations, `MPI_WIN_LOCK` is used where the fully-fledged server uses `MPI_WIN_LOCK_ALL`.

Server Failures

Ideally, a server or an I/O node that fails should not bring down the service. We would like such a server to rejoin the storage system when fixed. The observations are as follows:

Abort failure: Our experiences with aborting are presented in Table 6.1. OpenMPI aborts the job if any subset of `MPI_COMM_WORLD` is aborted. The job survives in MPICH and MVAPICH, and communications between the survivors (`healthy_comm`) proceed without issue. Eager sends to deceased peers complete in both MPICH and MVAPICH. All other communications trap the survivor in the progress engine in MVAPICH. In MPICH we qualify the behaviour as *Undefined* because it varies. In most cases, the communication returns immediately with an error message; this is the ideal case. This behaviour is observed for two-sided, collectives, and even one-sided, except for `MPI_WIN_UNLOCK` which gets blocked forever. In a few cases, receive calls get blocked; and so do send calls for Rendezvous-size messages.

Crash failure: In the case of crash failure simulations in a process, all three MPI implementations simply kill the job.

Table 6.1: Behaviours in case of isolated server abortion

	OpenMPI	**MPICH**	**MVAPICH**
No communication	WJA	GCE	TF
Communication over `healthy_comm`			
Any communication	N/A	Success	Success
Communication over `MPI_COMM_WORLD`			
2-sided; Eager, survivor is sender	N/A	SSS+GCE	SSS+TF
2-sided; Rendezvous, survivor is sender	N/A	Undefined	TC
2-sided; survivor is receiver	N/A	Undefined	TC
1-sided	N/A	Undefined	TC
Collectives	N/A	Undefined	TC

***WJA**: Whole job abortion; **GCE**: Graceful continuation and exit; **TF**: Trapped in `MPI_FINALIZE`;*
***TC**: Trapped in communication; **SSS**: Sender-side success*

Client Failures

The experiments in this subsection have been done only with OpenMPI and MPICH because we have not been able to successfully use port and connection-disconnection functions in MVAPICH. We run three servers on three nodes and a single client on a fourth node. As a reminder, each client shares an intercommunicator `connect_comm` and an intracommunicator `io_comm` with the storage system. We realize that both crash and abort failures in client jobs produce the same behaviours in the storage job. The results are presented in Table 6.2.

In both OpenMPI and MPICH, the storage job survives client job failure (crashes or abortions). No communication attempt with a deceased client brings down the storage job. The communication behaviours are similar for both the intercommunicator and the intracommunicator. In general, in OpenMPI, except for Eager sends, the storage processes get trapped in any explicitly or implicitly communicating routine. Except for Rendezvous sends and `MPI_WIN_UNLOCK`, where it gets trapped, MPICH returns an error message, and the execution continues gracefully. The immediate return and error messages allow the storage job to free resources in a timely fashion and to detect the failed client without custom additional mechanisms.

Table 6.2: Storage job behaviours after client job failure

	OpenMPI	**MPICH**
No communication	TCD	GCE
Internal storage job communications		
Any communication	Success+TCD	Success+GCE
Communication over `io_comm` and `connect_comm`		
2-sided; Eager, server is sender	SSS+TCD	SSS+GCE
2-sided; Rendezvous, server is sender	TC	TC
2-sided; server is receiver	TC	ER+GCE
1-sided (only over `io_comm`)	TC	TWU
Collectives	TC	ER+GCE

***GCE**: Graceful continuation and exit; **TCD**: Trapped in `MPI_ COMM_ DISCONNECT`; **TC**: Trapped in communication; **TWU**: Trapped in `MPI_ WIN_ UNLOCK`; **SSS**: Sender-side success; **ER**: Error return*

6.4.6 Object Limits

Each client group leads to the creation of two explicit communicators in each server process. Depending on the implementation, a third implicit communicator might be created for the one-sided communication window. Furthermore, in large systems, servers will have to maintain a very large number of handles to support nonblocking two-sided operations. The same is true for derived datatypes. Noncontiguous I/O operations require unique hindexed types for each I/O. In fact, an I/O transfer from a server has to resort to two separate derived datatypes because the non contiguity layout at the source is different from that of the target. In summary, we estimate that a server process needs 3 communicators (including the implicit one-sided window one if required), 1 one-sided window, 3 derived datatypes and 2 pending nonblocking point-to-point to service a single instance of any kind of I/O to a client. We verify through emulation tests whether a server can service a million process client job by creating 3,000,000 communicators, 1,000,000 one-sided windows, 2,000,000 nonblocking posted point-to-point and 3,000,000 hindexed derived datatypes. Each category of object is created in a separate test. The results are presented in Table 6.3. To detect resource exhaustion limits, we run each test on both systems C1 and C2, each time with a single process per node; even for the client job.

In addition to the observed limits, we report how each MPI implementation behaves after the limit is reached. For the point-to-point tests, the limit is determined by whichever of `MPI_ISEND` or `MPI_IRECV` fails first. In Table 6.3, we omit the cluster name in the result when the implementation behaves similarly on both. In the observations, "Resource exhaustion" might include situations related to pinning or mapped memory, for instance.

All three implementations have a hard limit for the number of communicators (Table 6.3). The limit seems to be a design choice because it does not depend on the amount of memory available on the node. For one-sided windows, OpenMPI succeeds in creating all the required objects on C2 but hits a limit on C1. One can notice that OpenMPI and MVAPICH can create more one-sided windows than communicators. Derived datatypes and point-to-points seem to be limited only by available memory. OpenMPI tends to block forever in both by-design and resource-imposed limits. MVAPICH either continues gracefully or crashes after

Table 6.3: Object limits

	OpenMPI	**MPICH**	**MVAPICH**
Communicators	65532+UB	2045+ER +GCE	2018+ER +GCE
1-sided windows	65532+UB on C1; NL on C2	2045+ER +GCE	2042+ER +GCE
Derived datatypes	NL	NL	RE+ER +GCE
Pending nonblocking 2-sided	RE+crash on C1; NL on C2	NL	RE+crash

***NL**: No limit; **UB**: Unlimited blocking; **ER**: Error return; **RE**: Resource exhaustion;*
***GCE**: Graceful continuation and exit*

returning an error code. MPICH has successfully created all the derived datatypes and non-blocking communications required to service a million clients. However, in order to observe its behaviour in situations of resource exhaustion, we increased substantially the number of objects. We observe that it behaves similarly to MVAPICH when resources are exhausted. Resource exhaustion-induced limits are not an issue per se; it is how the MPI implementation reacts to them that can make a difference, especially for the required persistence of the service.

6.5 Some Missing Scalability Features

In this section, we highlight a number of problem areas and formulate some recommendations for the MPI forum and MPI implementors. While some of these recommendations follow from our desire to use MPI in a non traditional setting, this does not preclude the usefulness of our recommendations for more mainstream MPI applications. Furthermore, we believe that as these applications evolve to support the fundamentally different environment presented by future exascale systems, some of the features described in this section will be required for all HPC software domains.

6.5.1 Fault Tolerance

Enforcing MPI_ERRORS_RETURN

For most contemporary HPC applications, the reasonable action in case of failures is to restart the job. However, when failed components can be reconstructed (for example from a replica), restarting is not always the best solution because other applications might depend on the same service. The issue of MPI jobs surviving the crash of a subset of `MPI_COMM_WORLD` has been previously studied [23]; recent similar proposals [9, 42] were also put forth during the standardization efforts of MPI-3.0. Unfortunately, none of these proposals made it into the standard. However, even without any change to the current specification, by simply provinding a standard-compliant implementation of `MPI_ERRORS_RETURN`, MPI implementations can already enable various workarounds to keep jobs alive after an isolated crash. When those crashes occur, and mechanisms are put in place to detect deceased processes, new healthy subsets of the surviving processes can be built incrementally from `MPI_COMM_SELF` by using bottom-up approaches similar to the non-collective communicator creation described in [20]. While the optimal fault-tolerance strategy at extreme scale is still being debated, we urge the research community to *provide some mechanism to continue MPI functionality in the presence of failures*, given the already-large demand for this capability [9].

A Plea for a More Cancellation-friendly MPI

From crash-resilient jobs [9, 23, 42] to MPI rank replication on spare hardware [24], fault-tolerance in MPI has gained more momentum recently. Checkpoint-recovery [43] which is the most widespread fault-tolerance approach in MPI is even offered in many MPI implementations. However, as stated in [34], the optimal fault-tolerance strategy at extreme scales is still not clear. In fact, even if MPI was to become fault-tolerant in the most idealistic way, programs and libraries built with MPI might still have their own additional requirements for resiliency. On top of any integrated fault-tolerance mechanism, the approach of MPI features as resiliency primitives should be encouraged to allow MPI applications, libraries and frameworks to easily realize their own fault-tolerance needs. The second of the three fault management concepts mentioned in [9] aligns with this suggestion; that concept advocates a flexible API meant to build fault-tolerant models as external libraries.

Cancellation could be a very useful resiliency primitive. Most large-scale filesystems such as PVFS2 [60] or Lustre [51] support request timeout. The same goes for most network technologies that could be used as transport for those systems. For instance BSD sockets and InfiniBand support communications with timeout. With respect to MPI, the third of the three fault management concepts advocated in [9] recommends the absence of indefinite time deadlock or block in case of failure. While timeout-enabled MPI communications will certainly find substantial adoption, cancellation could be even more useful. Cancellation is an even more fundamental primitive than timeout; and as such, it gives a bit more flexibility. First, timeout can easily be built on top of cancellation. Second, cancellation does allow a timed communication to be killed before any associated timeout. An example of use case is the situation where two redundant resources are solicited for the same operation; and the first completed operation renders the second one useless. Cancellation is a crucial mechanism for reclaiming resources when faults or other exceptional circumstances arise.

While MPI-3.0 extended the use of request objects (for example, for nonblocking collectives), it is still erroneous to issue `MPI_CANCEL` on any native nonblocking MPI function not associated with two-sided communications. Cancellation for network operations is widely known to be challenging. However, we showed in Section 6.4.4 that cancellation does not have to be perfect to be useful. MPI does offer means of cancelling custom functions built with generalized requests; but this feature is definitely less powerful compared with natively cancellable functions. In fact, generalized requests cannot transform non-cancellable existing MPI functionalities into cancellable ones. *Efforts should be invested in making most, if not all, MPI routines cancellable.*

6.5.2 Generalization of Nonblocking Operations

The current connection-disconnection handling at server side is non-scalable (Listing 6.3, Listing 6.4). The problem is not the connection or RPC listener threads being the single entry points for those respective procedures; the issue is the blocking nature of the involved non-communicating MPI collective routines. Currently, if server S_i is connecting or disconnecting the set $\{c_0, c_1, ..., c_n\}$ of connection objects, no server S_j can connect or disconnect a different set of connection object without potentially introducing a hazard. More importantly, no server S_j can connect or disconnect the elements of the exact same set of connection objects in a different order without introducing a hazard. Thus, connection and disconnection are not only serialized, they must occur in the exact same order on all the servers. That order, in the case of connections, is imposed by the root process specified in `MPI_COMM_ACCEPT`. In the case of disconnections, we impose it by having the root of any client group contact the exact same server every time (line 5 of Listing 6.3). Imposing the exact same server as the root to contact for those procedures hurts our resiliency, load balancing and scalability efforts which allow a client to retry any RPC with an alternative server node when one is not being responsive. Currently, the root server is a single point of failure in the service because, unlike the other servers, it does not have any strictly equivalent replica.

To understand why we have to resort to serialization and ordering of the connections and disconnections, let us assume the absence of those constraints and analyze how the service as a whole behaves as a consequence. We use disconnection for the analysis. We define:

- n as the number of distinct clients simultaneously asking for disconnection.
- w_i as the maximum number of disconnection work items that could be simultaneously handled by the server i.
- t_i as the number of CPU cores on the server i. We avoid oversubscribing the servers; so the upcoming analysis assumes that each server can run more than two hardware threads simultaneously.
- S as the set of servers.
- D_i as the set of connection objects being simultaneously disconnected by the server node i.

With the blocking routines offered by the current MPI specification for the disconnection procedures, since two threads are dedicated to connection and RPC listening, we obviously have

$$\forall\ i \in S,\ w_i = t_i - 2 \tag{6.1}$$

The situation modelled by Equation 6.2 results in the complete deadlock of the service as a whole; along with all the clients which had already started their disconnection procedure. $|D_i \cap D_j| = 0$ simply means that servers i and j are waiting for each other in collective calls over sets of mutually exclusive service-level disconnections. In concrete terms, they all block in the respective first collective call encountered in the disconnection procedure (line 6 of Listing 6.4). Then $(|D_i| = w_i) \wedge (|D_j| = w_j)$ means that all the threads of each of the servers are already occupied by the blocking server-level disconnection work items. There is no worker left for any of the two servers to, by chance, pick a work item that could transform $|D_i \cap D_j| = 0$ into $|D_i \cap D_j| > 0$.

$$\exists\ \{i, j\} \subset S \textit{ such that } (|D_i| = w_i) \wedge (|D_j| = w_j) \wedge |D_i \cap D_j| = 0 \tag{6.2}$$

The aforementioned deadlock is guaranteed to be avoided only if the number of simultaneous disconnection requests is strictly less than the sum of number of work items that any two servers can handle in parallel (Equation 6.3).

$$\forall\ \{i, j\} \subset S \textit{ such that } (|D_i| = w_i) \wedge (|D_j| = w_j),\ \boldsymbol{n < w_i + w_j} \implies |D_i \cap D_j| \geq 1 \tag{6.3}$$

$$\boldsymbol{\forall\ \{i, j\} \subset S,\ n < w_i + w_j} \tag{6.4}$$

Equation 6.3 states that a standstill will always ***eventually*** be broken as long as we guarantee $\boldsymbol{n < w_i + w_j}$. Intuitively, Equation 6.3 means that as long as we guarantee $\boldsymbol{n < w_i + w_j}$, if $|D_i| = w_i$, any existing standstill, due to D_i and D_j being disjoint, will be broken at the latest by the time $|D_j| = w_j$; and vice versa. In summary, Equation 6.4 is the absolute safety condition. In fact, as per Equation 6.1, Equation 6.4 leads to Equation 6.5; meaning that no more than 11 simultaneous clients are allowed on our 8-CPU core per node systems if deadlock is to be deterministically avoided. This number of clients is obviously far too small. However, processor core count per node in nowadays' systems is still limited; and the future might not promise substantially higher core counts because of hardware-level contention. Consequently, enforcing Equation 6.4 is not a viable option for an extreme-scale service. In fact, it is perfectly possible for a single multi-million process MPI job to open thousands of I/O connections which will all be requested for termination right before `MPI_FINALIZE`; leading to a massive amount of simultaneous disconnection requests sent to the service. As a background information, `MPI_FINALIZE` is the routine that is called after all MPI activities are ended in an MPI job.

$$\forall\ \{i, j\} \subset S,\ n < t_i + t_j - 4 \tag{6.5}$$

One can notice that ***no amount of CPU cores per server node can solve the problem created by Equation 6.2***. The opposite of Equation 6.5 is expressed by Equation 6.6 which is impossible to overcome. In fact, assuming that Equation 6.6 is already true, every time t_i or t_j is increased by 1, one just needs to increase n by 1 to maintain the deadlock condition.

$$\exists\ \{i, j\} \subset S,\ \textit{such that}\ n \geq t_i + t_j - 4 \tag{6.6}$$

Assuming the existence of nonblocking versions of some of the functions involved in the disconnection procedures, consider how the hypothetical approach of Listing 6.5 voids all the aforementioned problems. The disconnection work could be executed by resorting to the nonblocking collective cleanup functions on lines 7, 8, 9 of Listing 6.5. Then, on lines 10-12, a completion work is enqueued with the nonblocking handler at lines 14-28 of Listing 6.5. We emphasize that by the time the disconnection is initiated, the clients associated with the connection object must have already completed all their I/O; meaning that no one-sided communication is still in flight. As a result, `MPI_WIN_UNLOCK_ALL` will return immediately at line 6. `clean_zombie_receives` at line 5 will also return without wait because it has local semantics. With the possibility of handling the disconnection in a nonblocking way, even a single thread can progress any number of disconnections to completion. In fact, Equation 6.1 is now replaced by Equation 6.7; leading to Equation 6.4 always being true for any number of clients.

$$\forall\ i \in S,\ w_i = \infty \tag{6.7}$$

In general, blocking routines can be a hindrance to both performance and scalability. In some cases, concurrent requests are required in order to extract maximum hardware efficiency. At the same time, not all large HPC systems support the creation of an unlimited number of threads; and on systems that do, thread resource consumption typically prohibits creating a large number of threads. Having a thousand blocking routines pending requires no less than a thousand threads. In comparison, a threadpool of just a very few threads can do the same job if nonblocking routines are available. In fact it is useful to re-emphasize that, as shown

```
server_idisconnect_work_handler(work_t* work)
{
  MPI_Request reqs[3];
  conn_obj = get_conn_obj_from_work_item(work);
  clean_zombie_receives(conn_obj);//This has local semantics
  MPI_Win_unlock_all(conn_obj->win); //Will return immediately

  MPI_Win_ifree(&conn_obj->win, &req[0]); /*nonblocking version of MPI_Win_free*/
  MPI_Comm_ifree(&conn_obj->io_comm, &req[1]); /*nonblocking version of
    MPI_Comm_free*/
  MPI_Comm_idisconnect(&conn_obj->connect_comm, &req[2]); /*nonblocking version
    of MPI_Comm_disconnect*/
  work->handler = server_idisconnect_completion_work_handler;
  update_work_args_with_reqs(work, reqs);
  enqueue_work(work);
}
server_idisconnect_completion_work_handler(work_t* work)
{
  MPI_Request reqs[3]; int flag; MPI_Status statuses[3];
  fill_reqs_from_work_item(work, reqs);
  MPI_Testall(3, reqs, &flag, statuses);
  if(flag)
  {
    conn_obj = get_conn_obj_from_work_item(work);
    free(conn_obj);
    free(work);
    return;
  }
  update_work_args_with_reqs(work, reqs);
  enqueue_work(work);
}
```

Listing 6.5: Server-side nonblocking disconnection code snippet

by the previous disconnection deadlock hazard analysis, ***a higher degree of parallelism is***

not an alternative to the provision of nonblocking routines and vice versa. Both concepts do offer overlapping benefits but are not interchangeable. Furthermore, any blocking operation can be made nonblocking as long as consistency constraints are respected. Plus, by keeping blocking versions of potentially problematic nonblocking operations, users always have handy a safe alternative API to accomplish the same task in situations where consistency can be difficult to reason about. Similarly, a user can elect to use the blocking version of a functionality if a fast reaction is required.

Nonblocking operations seem to be a necessary condition for efficient cancellation as well. If cancellation was possible at all with blocking operations, two threads would be required for it to be effective on any single routine; one for the blocking call, and a second one to issue the cancellation. The `MPI_TEST` family of routines coupled with nonblocking versions of most routines would open up multiple possibilities for algorithms, programming paradigms (e.g. event-driven coding), performance and scalability decisions in code running on top of MPI. *We recommend continuing the effort to extend the set of nonblocking functions in MPI.*

6.5.3 Object Limits and Resource Exhaustion

HPC compute jobs rarely need thousands of communicators or RMA windows. The situation could be very different for persistent distributed services. Similarly, derived datatypes and nonblocking operations might exist in large quantities in large compute jobs but their numbers are usually reasonable in each process of the job. As shown by our design, any single process in persistent services can manage very large quantities of these objects at large scales. As a consequence, *by-design limits should be seriously lifted up to allow a broader set of use cases and to get MPI implementations ready for extreme scale uses.* Even at extreme scales, most programs will be naturally bound in many respects by architecture limits, such as word or pointer sizes in C. Thus, more natural by-design limits could be `INT_MAX`, `UINT_MAX` or `LONG_MAX`, `ULONG_MAX` for instance.

Furthermore, resource exhaustion is one of the most trivial expectations in very large programs. As a result, it is planned for and is therefore rarely an unmanageable issue. In fact, our storage service has a robust flow control policy that makes clients wait and retry in situations of resource exhaustion. Such a policy which delays selected clients without disrupting the service for everybody is much better than a sudden crash which renders the service totally unavailable. Ideally, *MPI should provide upper layers with non-fatal methods of discovering resource exhaustion*; so that these upper layers could decide and trigger workarounds if possible.

6.6 Summary

Portability and high performance are two attractive traits of the message passing interface. As MPI runs on top of the network fabric of its hosts, the aforementioned traits allow programmers to uniformly and efficiently target a very disparate set of supercomputers, each with its own architecture and network API. With the same concerns of portability and performance, we implement in MPI the network layer of a resilient, scalability-conscious distributed HPC storage system. We notice that a few features introduced in the recent MPI-3.0 specification substantially facilitate the mapping of the service semantics to MPI. Noteworthy are the request-based one-sided communication routines and dynamic windows. These features actually help in fulfilling key scalability design strategies. However, in certain areas, workarounds and design concessions were needed. The service exhibits use cases where nonblocking versions of non-communicating MPI routines would make a sound difference for performance, scalability and even safety. Another scalability issue created by the adoption of MPI resides in the runtime limit imposed on certain objects such as communicators. Furthermore, unlike regular MPI jobs, services are persistent and less tolerant to unplanned termination. As a consequence, the fault handling of MPI is also a major issue about which we provide analyses and recommendations. Finally, we provide an analysis of the suitability of `MPI_CANCEL` for the service design and discuss how it can be generalized to allow custom application or service-level resiliency design.

Chapter 7

Conclusion and Future Work

Communication is a central aspect of achieving high performance in supercomputing. In fact, a fraction of the data manipulated by the CPUs of supercomputers must be accessed via communications to remote nodes. How much wait occurs in the HPC application due to data dependency and data access is thus determined by the performance of the communication subsystem; a mandate that essentially falls on MPI in contemporary supercomputers. In this dissertation, we propose novel features or solutions to important aspects of MPI in order to avoid long CPU involvement in communication activities to the detriment of computation; and to avoid unintended resource idling. The proposed solutions, concepts or redesigns are scalability-driven in the sense that an emphasis is put on the avoidance of latency propagation and its cumulative negative effects that grow with scale. With the same scalability concern in mind, an emphasis is also put on memory consumption growth pattern. Optimized and scalable communications, as discussed in Chapter 3, Chapter 4 and Chapter 5, allows MPI jobs to reach larger and larger scales. However, once those scales are reached, other issues such as resource exhaustion and failure management must be provisioned for. Chapter 6 covers those issues by analyzing the ability to run HPC applications to completion in situations of high demand and partial failure. The analysis performed in Chapter 6 leads to resiliency and resource management proposals.

7.1 Summary of Findings

It can be shown that a dedicated thread or CPU core can efficiently overcome all the independent progress issues for large message transfers performed with the nonblocking Rendezvous protocol used in MPI middleware. Though, CPU core dedication for an entity that is not a worker process is usually both a luxury and a counterintuitive move for the end-user who invariably correlates higher performance and maximum parallelism. In this situation where CPU core dedication is not an option, the other viable alternative is to resort to the interrupt-based threading mechanism to get around the need for tight polling. However, the interrupt-threading approach deals an undue penalty to the applications, especially for blocking calls, as well as for nonblocking calls which always progress messages without additional help. The undue degradation caused by the interrupt-thread approach increases when the scale grows. Its memory behaviour also presents a scalability issue as its footprint grows quickly with the number of MPI processes hosted in each node. In order to avoid those issues, we propose a new approach to the Rendezvous message progression in Chapter 3. In presence of RDMA, we notice that efficient Rendezvous-backed message progression only requires small amounts of CPU cycles to fulfill the data transfer. Thus, we create a Parasite Execution Flow (PEF) that we sneak into an existing application thread to periodically steal those small amounts of CPU cycles for the purpose of checking Rendezvous control message arrival and trigger data transfers. On top of being as efficient as the interrupt-thread method, the PEF approach has an insignificant memory footprint and does not present the aforementioned scalability issues. Furthermore, PEF is scenario-conscious and operates only when required. As a result, it does not inflict any unjustified overhead to the application.

As shown with the Rendezvous protocol, nonblocking communication fulfilling can suffer a lack of autonomous message progression even with the presence of RDMA-enabled network device. Nevertheless, compared to blocking communications, nonblocking communications are such an important performance improving strategy that the HPC community strives to generalize their use. However, those nonblocking communications tend to create and increase backlogs of messages. As a result, MPI middleware must deal with intense message queue processing. As job sizes increase, message queues grow as well and can become slow to process; slowing in turn the communication that they support. We presented in Chapter 4 of this dissertation a new message queue tailored for the scalability needs of MPI. Adapted to the specific operations encountered in MPI message queue processing, the new 4D container that we put forth proved faster and far more memory-efficient than more generic search-directed data structures like binary trees. The two message queue architectures with the lowest memory footprint and lowest CPU consumption respectively have a linked list-based and an array-based designs. Assessment against those two reference architectures show that the 4D design is not only faster at scale than the linked list, but it can even be more memory-efficient. On very large systems, the array-based design can become impractical for consuming a large percentage of the physical memory available for each process. As a consequence, unlike the 4D approach proposed in this dissertation, the array-based queue does not allow MPI to realistically run jobs on supercomputers at petascale and beyond. The 4D message queue fulfills a few goals. It ensures that any linear traversal occurring during the searches is kept short. As a consequence, larger queue searches do not suffer noticeable degradation. Then, by narrowing down the search region along each dimension, the 4D message queue can determine and skip altogether very large queue portions which are guaranteed not to contain the sought item. Another major search optimization brought by the 4D message queue is the early detection of unfruitful searches in order to prevent useless traversals. With respect to memory behaviour, the 4D message queue allocates small structural objects whose memory footprints are amortized when the queue size grows. We built the message queue over strictly increasing intervals of communicator sizes. In a given interval, the memory consumption growth rate decreases when the communicator size grows. From a smaller interval to a larger one, the aforementioned growth rate flattens even further; leading to an overall scalable pattern of memory consumption.

The large amount of available MPI-based HPC programs use two-sided and collective communications; and as such, they are prone to generating substantial message queue items. For bearing no concept of reception, one-sided communications intrinsically do not queue messages even though they are nonblocking. One-sided communications are further shielded against the inter-CPU interactions inherent to communications such as those fulfilled with the Rendezvous protocol. In particular, one-sided communications embody an HPC data transfer paradigm that depends less on the availability of remote CPUs; making them suitable for latency propagation avoidance. However, the MPI one-sided communication model suffers from the blocking nature of its unavoidable synchronization routines. By proposing an entirely nonblocking approach to these synchronizations, we effectively realize the latency propagation reduction that is expected in theory from one-sided communications. The epochs enclosed by the synchronizations are no longer serialized. The MPI progress engine is empowered to concurrently schedule groups of one-sided communications belonging to the same window. The resulting out-of-order communication completion shrinks the overall duration of sets of one-sided communications; leading to an overall shorter execution for RMA-based HPC jobs. The improvements that we brought with the nonblocking synchronizations also fix for the first time a set of inefficiency patterns that were documented for MPI one-sided communications since MPI-2.0. Additionally, we discovered and documented a new inefficiency pattern during this work; and show that its fix resides in nonblocking synchronizations as well. The RMA proposal in Chapter 5 does not just resolve performance and scalability issues; it also makes it possible to express in MPI new dynamic unstructured communication patterns that are inefficient in the current MPI-3.0 specification.

Finally, the evolution of MPI to adapt to changing HPC realities is partially fuelled by real life use cases which reveal and back the need for supercomputing features required to achieve higher performance or reach higher scales. Thus, one of the major contributions of Chapter 6 is to uncover forward-looking MPI requirements that mainstream HPC programs cannot yet reveal. We used MPI for the communication substrate of an extreme-scale distributed storage system. The service differs from regular HPC programs in a number of ways. First, it does not complete; it runs continuously. Second, the service along with its collection of clients makes up a heterogeneous mix of unrelated and loosely coupled applications. As a result, our experience in Chapter 6 uses MPI in a cross-application setting where some jobs are created and terminated dynamically while another program, the service, is persistent. In comparison, regular HPC jobs are typically standalone applications. Furthermore, the processes of the service are expected to be hot-replaceable; meaning that MPI is expected to keep the storage job alive when isolated nodes become faulty. In comparison, typical HPC applications do not dynamically adapt to variable number of computing nodes for the purpose of resiliency. With respect to scalability, a large HPC job can create aggregated large numbers of communicators or RMA windows; but these objects exist in moderate numbers in each process of the job. In the storage service, each node can potentially interact with any number of clients. Thus, large numbers of the aforementioned objects can exist in any single process of the service; straining MPI to extents usually not reached by regular HPC applications. The unusual conditions created by the storage service led to important observations about how MPI implementations behave in situation of failure, resource exhaustion and massive per-process communication. All those situations represent challenges that MPI has to face in its evolution for exascale supercomputers. As solution proposals, we show that communication cancellation can be an important primitive for custom resiliency policy for software layers running on top of MPI. Then, we show that nonblocking approaches can be beneficial for non-communicating MPI routines as well. Nonblocking approaches are demonstrably a scalability strategy; but we show in the case of non-communicating routines that they can also prevent execution hazards. We finally put forth error communication from MPI to upper layers as a systematic resource exhaustion management strategy at extreme scale. This last recommendation contrasts with the current disparate policies that lead to sudden death of the HPC job when object limits are passed in various MPI implementations.

Each of the proposals made in this dissertation is implemented so as to show via experimentation that the ideas put forth lead to the promised improvements. We would like to reemphasize that the improvements claimed in this research owe to the concepts; not the implementations. In Chapter 3, our proposed Rendezvous protocol enhancement is implemented with signal delivery which is a heavyweight mechanism with a noticeable latency. The improvements that we reported in spite of the use of signal delivery show that the concept per se bears sound promises for performance and resource consumption gain. Chapter 4 makes a clear distinction between the theoretical performance analysis (Section 4.4) and experimental evaluation (Section 4.6). The theoretical performance analysis, which is implementation-independent, describes the improvement pattern expected from a purely conceptual point of view. Finally, in Chapter 5, we show in Section 5.4 how nonblocking RMA synchronizations can substantially decrease contention, avoid epoch serialization, avoid latency propagation and mitigate peer delays by overlapping communication with delay or computation with delay. These improvements were discussed either analytically or with use cases that are all implementation-independent. We remind that Chapter 6 does not put forth performance-oriented proposals; instead, it reveals MPI behaviours that can be improved and formulates recommendations.

7.2 Future Work

Asynchronous Message Progression without Dedicated Threads

We mentioned in Section 3.2.2 the parameters *ppef_period*, *ppef_phase*, *ppef_freq_decay*, *ppef_max_turns_per_req* meant to control the period, the phase, the frequency decay and the maximum number of AEF/PEF transitions before PEF gives up watching for Rendezvous control message arrival. The tuning space defined by these parameters is very large. As a future work, we intend to come up with heuristics for a default generally optimal sets of values for these parameters.

Furthermore, we mentioned that a sufficient condition for effective message progression is the availability of a carousel, a trigger and a watchdog. In the proposal of Chapter 3, PEF fulfills both the watchdog and the trigger. For a given communication, once the trigger is fulfilled, PEF stops or dies; meaning that the watchdog is the requirement that could force PEF to intervene more than once for a single communication. Both the watchdog and the trigger require very little CPU activity. In fact, a carefully designed watchdog could boil down to simply checking the value of an integer. Nevertheless, whenever PEF preempts the default execution flow of the thread that it silently rides, the whole state of the thread must be saved, as per the default mechanism used by the OS. As the watchdog activity can be required several times for a single communication, the cost of that in-thread context-switch could be incurred several times. Resorting to a mechanism as heavy as signal delivery to check the state of an integer obviously leaves a lot of room for improvement. As a future work, we seek to replace signal delivery with a mechanism that could actually be made so lightweight that

PEF would become almost entirely penalty-free. We rule out compile-time alteration because of the possible use of system calls and third party libraries whose source codes might not be available. We plan to investigate how the OS itself can be altered to fulfill a lightweight PEF. Instead of saving the whole CPU context as well as the thread stack on each watchdog activity, a very limited number of select CPU registers could be manually saved. The preemption itself could be realized by designing a custom trap-like mechanism that would be deprived of all the automatic state saving that the OS usually does. Then if, from the value of the integer or flag, the watchdog does not infer the arrival of the control message, the instructions simply restore the manually saved limited set of registers and branch back to the regular execution of the thread. This whole watchdog activity can be coded in a few assembly instructions and execute very fast. If a watchdog activity detects control message arrival, then a heavy context-switch is performed once and the trigger is fulfilled. The whole proposal would still be network device-agnostic.

Alternatively, the watchdog activity can be totally offloaded to the OS and occur inside the kernel; and the application thread is interrupted only when the watchdog discovers the control message. The watchdog in this case would not poll; it would execute only once by simply being reactive to control message arrival. For this approach to work, each `MPI_IRECV` for which the problematic Rendezvous scenario occurs could notify the OS of the need to interrupt the application flow when the control message arrives. Since RDMA-enabled network devices tend to bypass the operating system for communications, a co-design of OS and network device driver is required for this last approach.

Scalable Message Queue

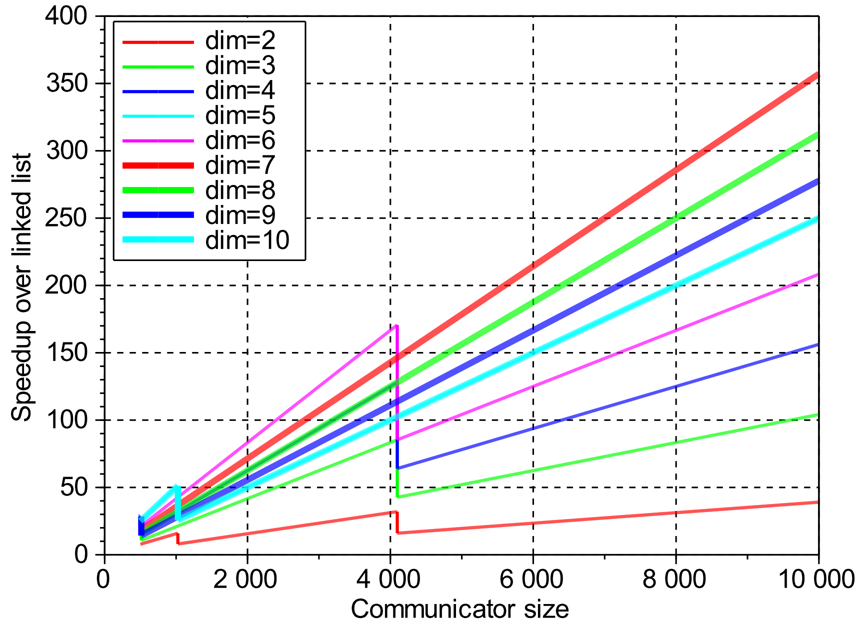
The 4D message queue has a fixed number of dimensions; and it varies *dimensionSpan* according to the communicator size. As a reminder, *dimensionSpan* is the maximum number of distinct coordinate values hosted by each dimension. One of the goals of the multidimensional approach is to keep chunks of linear traversal short by keeping *dimensionSpan* short. The 4D data structure does achieve that goal thanks to the very slow increase of *dimensionSpan* when the communicator size grows. However, it would be interesting to study the effect of setting an absolute cap on the length of linear traversals by fixing *dimensionSpan* to a specific value (for instance 16); and then increasing the number of dimensions dynamically as required by the communicator size.

We also observe in Figure 7.1 that the optimal message queue dimensionality for each communicator size can vary in a non-linear fashion. For instance, Figure 7.1(a) shows that a 10-dimension design is theoretically best for communicator sizes up to 1000. Then 6-dimension and 7-dimension queue designs are best respectively up to about 4000 and from 4000 onward. The same observation can be made for a larger range of communicator sizes on Figure 7.1(b). In fact, these non-linear changes in optimal dimensionality continue towards infinity. We plan to investigate this observation further and see how it translates into a concrete message queue architecture where both the dimensionality and *dimensionSpan* change dynamically to provide the best speed of processing for various communicator size intervals.

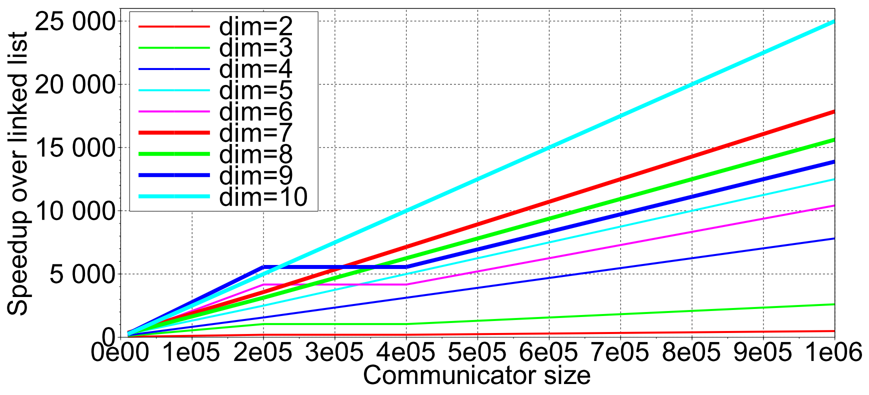
We are also in the process of observing the behaviour of MPI message queues, and the 4-dimensional message queue in particular, on Intel Xeon Phi processors [94]. These many-core CPUs have the particularity of not supporting out-of-order execution. Consequently, compared to mainstream CPUs, Xeon Phi processors have a different sensitivity to data access patterns, and possibly message queue architectures.

Nonblocking MPI One-Sided Communication Synchronization

We showed that RMA epochs can complete out-of-order even with the default behaviour that we designed for the progress engine. We also provide the MPI application programmer with various info object key-value pairs to boost the level of concurrent epoch progression that is deemed safe for his specific needs. However, the internal opening of RMA epochs at middleware level must still occur in a FIFO fashion. Removing that constraint introduces substantially more hazards at application level. At middleware level, it also adds quite some complexity. However, such an option should still be given to the bold MPI programmer, as it allows the progress engine to perform the most aggressive form of epoch progression. As future work, we intend to carefully analyze and define the semantics of a total out-of-order middleware-level epoch opening and progression; and then come up with a proposal that still guarantees



(a) Small range



(b) Large range

Figure 7.1: Theoretical optimal message queue dimensionality in terms of communicator sizes

correctness when certain application-level conditions are met.

One well-known HPC optimization consists of aggregating small distinct messages into a single data transfer in order to amortize the communication startup cost and make a better utilization of the bandwidth. This practice is termed *message coalescing* or *message aggregagtion*. In MPI one-sided communication, message aggregation is deferred to either the epoch-closing routine or to a ***flush*** call. Since blocking synchronizations force all the RMA data transfer to occur inside the epoch, no communication/computation overlapping is possible for coalesced messages. In fact, there is an unavoidable tradeoff between 1) reaping the benefit of message aggregation and incurring a communication/computation serialization and 2) avoiding the serialization and giving up the benefits of message aggregation. With nonblocking synchronizations, the deferral of the aggregation ceases to be an issue because the data transfer can occur outside the epoch. In particular, any transfer internally initiated in the nonblocking epoch-closing synchronization call necessarily occurs after the epoch is closed; allowing the application to still overlap the communication of the closed epoch with a subsequent activity. As a consequence, the aforementioned tradeoff disappears. With the negative side effects out of way, we intend to propose as future work a very flexible hint-based one-sided message aggregation. The hint, provided via info objects to the RMA window, could cover aspects such as message kinds, aggregation count or epoch-types.

Service Orientation and Provision for the Revealed Missing Scalability Features

We intend to work with both the MPI forum and implementers to ensure that MPI remains a driving force for future software and hardware architectures. We plan to pursue research on the possibility for service-oriented and multi-application HPC programs to leverage the power of MPI on nowadays' and future supercomputers. We are also currently working on nonblocking designs of non-communicating MPI routines which, like the functions used in the connection-disconnection of the storage service of Chapter 6, could lead to scalability issues that even higher CPU core counts could not work around.

Bibliography

[1] Argonne National Laboratory. Communicators and Context IDs. http://wiki.mcs.anl.gov/mpich2/index.php/Communicators_and_Context_IDs. Online; accessed 2013-12-14.

[2] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: a View from Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf, 2006. Technical Report UCB/EECS-2006-183. Online; accessed 2013-12-14.

[3] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff. MPI on Millions of Cores. *Parallel Processing Letters (PPL)*, 21(1):45–60, Mar. 2011.

[4] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk. The Importance of Non-Data-Communication Overheads in MPI. *International Journal of High Performance Computing Application (IJHPCA)*, 24(1):5–15, Feb. 2010.

[5] M. Banikazemi, R. Govindaraju, R. Blackmore, and D. Panda. Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation. In *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 183–190, 1999.

[6] B. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. Maccabe, and T. Hudson. The Portals 4.0.1 Network Programming Interface. http://www.cs.sandia.gov/Portals/portals401.pdf. Online; accessed 2013-12-06.

[7] 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.

[8] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick.

ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`, 2008. Online; accessed 2013-12-14.

[9] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra. An Evaluation of User-Level Failure Mitigation Support in MPI. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 193–203. Springer Berlin Heidelberg, 2012.

[10] D. Bonachea and J. Duell. Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations. *International Journal of High Performance Computing and Networking (IJHPCN)*, 1(1-3):91–99, Aug. 2004.

[11] 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.

[12] R. Brightwell, S. Goudy, and K. Underwood. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP)*, pages 175–183. IEEE Computer Society, 2005.

[13] R. Brightwell, S. P. Goudy, A. Rodrigues, and K. D. Underwood. Implications of Application Usage Characteristics for Collective Communication Offload. *International Journal of High Performance Computing and Networking (IJHPCN)*, 4(3/4):104–116, Aug. 2006.

[14] R. Brightwell and K. D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. In *Proceedings of the 2004 International Parallel and Distributed Processing Symposium (IPDPS)*, pages 183–, 2004.

[15] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 26:1–26:10. ACM, 2011.

[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] N. Desai, R. Bradshaw, A. Lusk, and E. Lusk. MPI Cluster System Software. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 277–286. Springer Berlin Heidelberg, 2004.

[19] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *Proceedings of the 2012 International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750, 2012.

[20] J. Dinan, S. Krishnamoorthy, P. Balaji, J. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu. Noncollective Communicator Creation in MPI. In *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 282–291. Springer Berlin Heidelberg, 2011.

[21] J. Dongarra. Visit to the National University for Defense Technology Changsha, China. `http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf`. Online; accessed 2013-12-13.

[22] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The International Exascale Software Project: A Call To Cooperative Action By the Global High-Performance Community. *International Journal of High Performance Computing Applications (IJHPCA)*, 23(4):309–322, Nov. 2009.

[23] G. E. Fagg and J. J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *International Journal of High Performance Computing Applications (IJHPCA)*, 18(3):353–361, 2004.

[24] K. Ferreira, J. Stearley, J. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2011.

[25] R. D. G. Burns and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of the 1994 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 379–386, 1994.

[26] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer Berlin Heidelberg, 2004.

[27] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 53:1–53:12. ACM, 2013.

[28] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, 2010.

[29] M. Gollcbiewski and J. L. Träff. 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.

[30] L. Graham, Richard, R. Brightwell, B. Barrett, G. Bosilca, and J. Pjesivac-Grbovic. An Evaluation of Open MPI's Matching Transport Layer on the Cray XT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 161–169. Springer Berlin Heidelberg, 2007.

[31] 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.

[32] R. E. Grant, M. J. Rashti, P. Balaji, and A. Afsahi. RDMA Capable iWARP over Datagrams. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 628–639, 2011.

[33] W. Gropp. MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin Heidelberg, 2012.

[34] W. Gropp and E. Lusk. Fault Tolerance in Message Passing Interface Programs. *International Journal of High Performance Computing Application (IJHPCA)*, 18(3):363–372, Aug. 2004.

[35] 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.

[36] S. Gutierrez, N. Hjelm, M. Venkata, and R. Graham. Performance Evaluation of Open MPI on Cray XE/XK Systems. In *Proceedings of the 2012 Annual Symposium on High-Performance Interconnects (HOTI)*, pages 40–47, 2012.

[37] Z. Haiyang and L. Qiaoyu. Red-Black Tree Used for Arranging Virtual Memory Area of Linux. In *Proceedings of the 2010 International Conference on Management and Service Science (MASS)*, pages 1–3, 2010.

[38] 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.

[39] 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.

[40] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards Efficient MapReduce Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume

5759 of *Lecture Notes in Computer Science*, pages 240–249. Springer Berlin Heidelberg, 2009.

[41] T. Hoefler and M. Snir. Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. In *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 345–355. Springer Berlin Heidelberg, 2011.

[42] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. Solt. Run-Through Stabilization: An MPI Proposal for Process Fault Tolerance. In *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 329–332. Springer Berlin Heidelberg, 2011.

[43] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[44] InfiniBand. `http://www.infinibandta.org/`. Online; accessed 2013-12-04.

[45] 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.

[46] Intel. Intel threading building blocks. `https://www.threadingbuildingblocks.org/`. Online; accessed 2013-12-14.

[47] 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 *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 68–76. Springer Berlin Heidelberg, 2004.

[48] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-sided Communication over InfiniBand. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 531–538, 2004.

[49] R. Keller and R. L. Graham. Characteristics of the Unexpected Message Queue of MPI Applications. In *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 179–188. Springer Berlin Heidelberg, 2010.

[50] K. Kharbas, D. Kim, T. Hoefler, and F. Mueller. Assessing HPC Failure Detectors for MPI Jobs. In *Proceedings of the 2012 Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 81–88, 2012.

[51] R. Knepper, S. Michael, W. Johnson, R. Henschel, and M. Link. The Lustre File System and 100 Gigabit Wide Area Networking: An Example Case from SC11. In *Proceedings of the 2012 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 260–267, 2012.

[52] 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.

[53] 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.

[54] M. Krishnan, J. Nieplocha, M. Blocksome, and B. Smith. Evaluation of Remote Memory Access Communication on the IBM Blue Gene/P Supercomputer. In *Proceedings of the 2008 International Conference on Parallel Processing (ICPP)*, pages 109–115, 2008.

[55] 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.

[56] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. 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.

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

[58] P. Lai, S. Sur, and D. Panda. Designing Truly One-sided MPI-2 RMA Intra-node Communication on Multi-core Systems. *Computer Science - Research and Development*, 25(1-2):3–14, 2010.

[59] C. Lameter. Shoot First and Stop the OS Noise. `https://www.kernel.org/doc/ols/2009/ols2009-pages-159-168.pdf`. Linux Foundation. Online; accessed 2014-02-12.

[60] R. Latham, R. Ross, and R. Thakur. Can MPI Be Used for Persistent Parallel Services? In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 275–284. Springer Berlin Heidelberg, 2006.

[61] 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.

[62] X. Lu, B. Wang, L. Zha, and Z. Xu. Can MPI Benefit Hadoop and MapReduce Applications? In *Proceedings of the 2011 International Conference on Parallel Processing Workshops (ICPPW)*, pages 371–379, 2011.

[63] M. Luo, S. Potluri, P. Lai, E. Mancini, H. Subramoni, K. Kandalla, S. Sur, and D. Panda. High Performance Design and Implementation of Nemesis Communication Layer for Two-Sided and One-Sided MPI Semantics in MVAPICH2. In *Proceedings of the 2010 International Conference on Parallel Processing Workshops (ICPPW)*, pages 377–386, 2010.

[64] E. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 2008.

[65] 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.

[66] Mellanox Technologies Inc. Why Compromise? A Discussion on RDMA versus Send/Receive and the Difference Between Interconnect and Application Semantics. `http://www.mellanox.com/pdf/whitepapers/WP_Why_Compromise_10_26_06.pdf`. White Paper. Online; accessed 2013-12-13.

[67] H. Miyazaki, Y. Kusano, H. Okano, T. Nakada, K. Seki, T. Shimizu, N. Shinjo, F. Shoji, A. Uno, and M. Kurokawa. K Computer: 8.162 PetaFLOPS Massively Parallel Scalar Supercomputer Built with over 548k Cores. In *Proceedings of the 2012 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 192–194, 2012.

[68] MPI Forum. The Message Passing Interface. `http://www.mpi-forum.org/`. Online; accessed 2013-12-04.

[69] MPICH: High Performance Portable MPI. `http://www.mpich.org/`. Online; accessed 2013-12-11.

[70] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[71] MVAPICH. `http://mvapich.cse.ohio-state.edu/`. Online; accessed 2013-12-11.

[72] Myrinet. `http://www.myri.com/`. Online; accessed 2013-12-13.

[73] Myrinet Express (MX): A High Performance, Low-level, Message-passing Interface for Myrinet. `http://www.myri.com/scs/MX/doc/mx.pdf`, 2003. Online; accessed 2013-12-17.

[74] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/resources/software/npb.html`. Online; accessed 2013-12-14.

[75] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Application (IJHPCA)*, 20(2):203–231, May 2006.

[76] J. Nieplocha, V. Tipparaju, and E. Apra. An Evaluation of Two Implementation Strategies for Optimizing One-Sided Atomic Reduction. In *Proceedings of the 2005 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215.2–, 2005.

[77] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling Parallel I/O Performance through I/O Delegate and Caching System. In *Proceedings of the 2008 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:12, 2008.

[78] OpenFabrics Alliance. `http://www.openfabrics.org/`. Online; accessed 2013-12-13.

[79] Open MPI. `http://www.open-mpi.org/`. Online; accessed 2013-12-11.

[80] 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.

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

[82] S. J. Plimpton and K. D. Devine. MapReduce in MPI for Large-scale Graph Algorithms. *Journal of Parallel Computing*, 37(9):610–632, Sept. 2011.

[83] S. Ramalingam, M. Hall, and C. Chen. Improving High-Performance Sparse Libraries Using Compiler-Assisted Specialization: A PETSc Case Study. In *Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 487–496. IEEE Computer Society, 2012.

[84] 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.

[85] 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.

[86] RDMA Consortium. `http://www.rdmaconsortium.org`. Online; accessed 2013-12-11.

[87] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. `http://tools.ietf.org/search/rfc5040`. Online; accessed 2014-01-13.

[88] J. M. Richter and C. Nasarre. *Windows via C/C++*. Microsoft Press, 5th edition, 2007.

[89] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda. Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand. In *Proceedings of the 2009 IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 380–387. IEEE Computer Society, 2009.

[90] 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.

[91] G. Santhanaraman, S. Narravula, and D. Panda. Designing Passive Synchronization for MPI-2 One-sided Communication to Maximize Overlap. In *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2008.

[92] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. Message Passing and Shared Address Space Parallelism on an SMP Cluster. *Parallel Computing*, 29(2):167 – 186, 2003.

[93] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 49–49, 2001.

[94] M. Si, Y. Ishikawa, and M. Tatagi. Direct MPI Library for Intel Xeon Phi Co-Processors. In *Proceedings of the 2013 IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 816–824, 2013.

[95] 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.

[96] 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.

[97] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling Remote Procedure Call for High-Performance Computing. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing (Cluster)*, 2013.

[98] T. Sterling. HPC in Phase Change: Towards a New Execution Model. In *High Performance Computing for Computational Science VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 31–31. Springer Berlin Heidelberg, 2011.

[99] H. Su, B. Gordon, S. Oral, and A. George. SCI Networking for Shared-memory Computing in UPC: Blueprints of the GASNet SCI Conduit. In *Proceedings of the 2004 Annual IEEE International Conference on Local Computer Networks (LCN)*, pages 718–725, 2004.

[100] S.-J. Sul and A. Tovchigrechko. Parallelizing BLAST and SOM Algorithms with MapReduce-MPI Library. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 481–489, 2011.

[101] Y. Sun, G. Zheng, L. Kale, T. Jones, and R. Olson. A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect. In *Proceedings of the 2012 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 751–762, 2012.

[102] S. Sur, H.-W. 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.

[103] N. Tanabe, A. Ohta, P. Waskito, and H. Nakajo. Network Interface Architecture for Scalable Message Queue Processing. In *Proceedings of the 2009 International Conference on Parallel and Distributed Systems (ICPADS)*, pages 268–275, 2009.

[104] 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.

[105] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Traff. Investigating High Performance RMA Interfaces for the MPI-3 Standard. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 293–300. IEEE Computer Society, 2009.

[106] Top500. `http://www.top500.org/`. Online; accessed 2013-12-04.

[107] B. Tourancheau and R. Westrelin. Support for MPI at the Network Interface Level. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 52–60. Springer Berlin Heidelberg, 2001.

[108] 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.

[109] K. Underwood and R. Brightwell. The Impact of MPI Queue Usage on Message Latency. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP)*, pages 152–160, 2004.

[110] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Proceedings of the 2005 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 96.2–. IEEE Computer Society, 2005.

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

[112] D. Wang and J. Liu. Peer-to-Peer Asynchronous Video Streaming using Skip List. In *2006 IEEE International Conference on Multimedia and Expo*, pages 1397–1400, July 2006.

[113] L. Xiao and T. Yu-An. TPL: A Data Layout Method for Reducing Rotational Latency of Modern Hard Disk Drive. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering (WCECS)*, volume 7, pages 336–340, 2009.

[114] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO)*, pages 24–32. ACM, 2007.

[115] X. Zhao, D. Buntinas, J. A. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. Toward Asynchronous and MPI-Interoperable Active Messages. In *Proceedings of the 2013 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 87–94, 2013.

[116] X. Zhao, G. Santhanaraman, and W. Gropp. Adaptive Strategy for One-Sided Communication in MPICH2. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 16–26. Springer Berlin Heidelberg, 2012.

[117] 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.

[118] J. A. Zounmevo and A. Afsahi. An Efficient MPI Message Queue Mechanism for Large-scale Jobs. In *Proceedings of the 2012 International Conference on Parallel and Distributed Systems (ICPADS)*, pages 464–471, 2012.

[119] J. A. Zounmevo and A. Afsahi. A Fast and Resource-conscious MPI Qessage Queue Mechanism for Large-scale Jobs. *Journal of Future Generation Computer Systems*, 30(0):265–290, 2014.

[120] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Using MPI in High-Performance Computing Services. In *Proceedings of the 2013 European MPI Users' Group Meeting (EuroMPI)*, pages 43–48. ACM, 2013.

[121] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Extreme-Scale Computing Services over MPI: Experiences, Observations and Features Proposal for Next Generation Message Passing. *International Journal of High Performance Computing Applications (IJHPCA)*, 2014. Under review.

[122] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi. Nonblocking Epochs for MPI One-Sided Communications. 2014. Submitted to the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC).

List of Publications

Journal Publications

J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Extreme-Scale Computing Services over MPI: Experiences, Observations and Features Proposal for Next Generation Message Passing. International Journal of High Performance Computing Applications (IJHPCA), 2014. Recommended for publication with minor revisions.

J. A. Zounmevo and A. Afsahi. A Fast and Resource-conscious MPI Message Queue Mechanism for Large-scale Jobs. Journal of Future Generation Computer Systems, 30(0):265290, 2014

Conference Publications

J. A. Zounmevo and A. Afsahi, Intra-Epoch Message Scheduling to Exploit Unused or Residual Overlapping Potential, Submitted to the 2014 Euro MPI/Asia Conference, 2014. Under Review.

J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi. Nonblocking Epochs for MPI One-Sided Communications. Submitted to the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2014. Under review.

J. Soumagne, D. Kimpe, **J. A. Zounmevo**, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling Remote Procedure Call for High-Performance Computing. In Proceedings of the 2013 IEEE International Conference on Cluster Computing (Cluster), 2013.

X. Zhao, D. Buntinas, **J. A. Zounmevo**, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. Toward Asynchronous and MPI-Interoperable Active Messages. In Proceedings of the 2013 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 8794, 2013.

J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Using MPI in High-Performance Computing Services. In Proceedings of the 2013 European MPI Users Group Meeting (EuroMPI), pages 4348. ACM, 2013.

J. A. Zounmevo and A. Afsahi. An Efficient MPI Message Queue Mechanism for Large scale Jobs. In Proceedings of the 2012 International Conference on Parallel and Distributed Systems (ICPADS), pages 464471, 2012.

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 542546, 2011.